

# DW\_fifo\_2c\_df

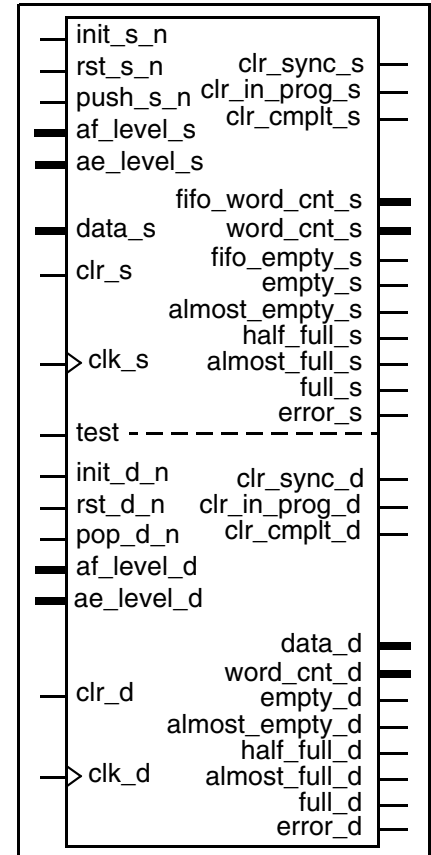
## Dual Independent Clock FIFO

Version, STAR, and myDesignWare Subscriptions: [IP Directory](#)

### Features and Benefits

- Pop interface caching (pre-fetching)
- Alternative pop cache implementations provided for optimum power savings
- Configurable pipelining of push and pop control/ data to accommodate synchronous RAMs
- Single clock cycle push and pop operations
- Fully registered synchronous status flag outputs
- Status flags provided from each clock domain
- Parameterized data width
- Parameterized RAM depth
- Parameterized number of synchronization stages in each clock domain
- Parameterized full-related and empty-related flag thresholds per clock domain
- Push error (overflow) and pop error (underflow) flags per clock domain
- Includes a low-power implementation that has power benefits from minPower optimization (for details, see [Table 1-3](#) on page 5)

### Revision History



### Description

DW\_fifo\_2c\_df is a dual independent clock FIFO consisting of DesignWare components DW\_fifocntl\_2c\_df (FIFO controller) and DW\_ram\_r\_w\_2c\_dff (dual-port synchronous RAM). Word caching (or pre-fetching) is performed in the pop interface to minimize latencies and allow for bursting of contiguous words. The caching depth is configurable.

Synchronous RAM that is supported can have one of the following architectures:

- Non re-timed write port and asynchronous read port
- Re-timed write port and asynchronous read port
- Non re-timed write port and synchronous read port with buffered read address and non-buffered read data

- Non re-timed write port and synchronous read port with non-buffered read address and buffered read data
- Non re-timed write port and synchronous read port with buffered read address and buffered read data
- Re-timed write port and synchronous read port with buffered read address and non-buffered read data
- Re-timed write port and synchronous read port with non-buffered read address and buffered read data
- Re-timed write port and synchronous read port with buffered read address and buffered read data

To accommodate the dual clock environment, parameters are provided to adjust the number of synchronization stages needed in both directions between the two clock domains.

To provide a clean reset environment, the FIFO contains reset logic that coordinates clearing of both clock domains in a controlled and orchestrated algorithm for localized resets operations (see Clearing FIFO section).



#### Note

Unless otherwise stated here on out, the term FIFO means the grouping of the RAM module and pre-fetching cache.

As of DesignWare versions F-2011.09-SP1 and later, the underlying component DW\_fifoclr\_2c\_df contains the DW\_reset\_sync that was enhanced to improve the clearing sequence. As a result, [Figure 1-11](#), [Figure 1-12](#) and [Figure 1-13](#) of this datasheet have been updated to reflect the change of behavior of the clearing sequence. This enhancement of DW\_reset\_sync does not impact the basic function of DW\_fifo\_2c\_df with respect to the data flow and status reporting before, during, and after the clearing sequence.

For more details about the changes to DW\_reset\_sync see [DW\\_reset\\_sync-datasheet](#).

**Table 1-1 Pin Description**

Pin Name	Width	Direction	Function
clk_s	1 bit	Input	Source domain clock
rst_s_n	1 bit	Input	Source domain asynchronous reset (active low)
init_s_n	1 bit	Input	Source domain synchronous reset (active low)
clr_s	1 bit	Input	Source domain clear RAM contents
ae_level_s	$\text{ceil}(\log_2[\text{ram\_depth}] + 1)$	Input	Source domain almost empty level for the <code>almost_empty_s</code> output (the number of words in the RAM at or below which the <code>almost_empty_s</code> flag is active) (see Note on <code>eff_depth</code> below table)
af_level_s	$\text{ceil}(\log_2[\text{ram\_depth}] + 1)$	Input	Source domain almost full level for the <code>almost_full_s</code> output (the number of empty memory locations in the RAM at which the <code>almost_full_s</code> flag is active)
push_s_n	1 bit	Input	Source domain push request (active low)

Table 1-1 Pin Description (Continued)

Pin Name	Width	Direction	Function
data_s	width	Input	Source domain push data
clr_sync_s	1 bit	Output	Source domain coordinated clear synchronized (reset pulse that goes source sequential logic)
clr_in_prog_s	1 bit	Output	Source domain clear in progress
clr_cmplt_s	1 bit	Output	Source domain clear complete (single <code>clk_s</code> cycle pulse)
fifo_word_cnt_s	$\text{ceil}(\log_2(\text{eff\_depth} + 1))$	Output	Source domain total word count in the RAM and cache NOTE: In the Width column, <code>eff_depth</code> is derived from <code>mem_mode</code> and <code>ram_depth</code> , as defined in Table 1-2.
word_cnt_s	$\text{ceil}(\log_2(\text{eff\_depth} + 1))$	Output	Source domain RAM word count NOTE: In the Width column, <code>eff_depth</code> is derived from <code>mem_mode</code> and <code>ram_depth</code> , as defined in Table 1-2.
fifo_empty_s	1 bit	Output	Source domain FIFO empty flag
empty_s	1 bit	Output	Source domain RAM empty flag
almost_empty_s	1 bit	Output	Source domain RAM almost empty flag (determined by <code>ae_level_s</code> input)
half_full_s	1 bit	Output	Source domain RAM half full flag
almost_full_s	1 bit	Output	Source domain RAM almost full flag (determined by <code>af_level_s</code> input)
full_s	1 bit	Output	Source domain RAM full flag
error_s	1 bit	Output	Source domain push error flag (overflow)
clk_d	1 bit	Input	Destination domain clock
rst_d_n	1 bit	Input	Destination domain asynchronous reset (active low)
init_d_n	1 bit	Input	Destination domain synchronous reset (active low)
clr_d	1 bit	Input	Destination domain clear RAM contents
ae_level_d	$\text{ceil}(\log_2[\text{eff\_depth}] + 1)$	Input	Destination domain almost empty level for the <code>almost_empty_d</code> output (the number of words in the FIFO at or below which the <code>almost_empty_d</code> flag is active) NOTE: In the Width column, <code>eff_depth</code> is derived from <code>mem_mode</code> and <code>ram_depth</code> , as defined in Table 1-2.

**Table 1-1 Pin Description (Continued)**

Pin Name	Width	Direction	Function
af_level_d	$\text{ceil}(\log_2[\text{eff\_depth}] + 1)$	Input	Destination domain almost full level for the <code>almost_full_d</code> output (the number of empty memory locations in the FIFO at which the <code>almost_full_d</code> flag is active) NOTE: In the Width column, <code>eff_depth</code> is derived from <code>mem_mode</code> and <code>ram_depth</code> , as defined in <a href="#">Table 1-2</a> .
pop_d_n	1 bit	Input	Destination domain pop request (active low)
clr_sync_d	1 bit	Output	Destination domain coordinated clear synchronized (reset pulse that goes to destination sequential logic)
clr_in_prog_d	1 bit	Output	Destination initiated clear in progress
clr_cmplt_d	1 bit	Output	Destination domain clear complete (single <code>clk_d</code> cycle pulse)
data_d	width	Output	Destination domain data to pop
word_cnt_d	$\text{ceil}(\log_2(\text{eff\_depth} + 1))$	Output	Destination domain total word count in the RAM and cache NOTE: In the Width column, <code>eff_depth</code> is derived from <code>mem_mode</code> and <code>ram_depth</code> , as defined in <a href="#">Table 1-2</a> .
empty_d	1 bit	Output	Destination domain empty flag
almost_empty_d	1 bit	Output	Destination domain FIFO almost empty flag (determined by <code>ae_level_d</code> input)
half_full_d	1 bit	Output	Destination domain FIFO half full flag
almost_full_d	1 bit	Output	Destination domain FIFO almost full flag
full_d	1 bit	Output	Destination domain FIFO full flag (determined by <code>af_level_d</code> input)
error_d	1 bit	Output	Destination domain error flag (under-run)
test	1 bit	Input	Scan test mode select

The width of some inputs is determined by `eff_depth` (effective depth), which is derived from the `mem_mode` and `ram_depth` parameters, as defined in [Table 1-2](#):

**Table 1-2 Effective Depth (`eff_depth`) of FIFO**

Effective Depth Based on <code>mem_mode</code> and <code>ram_depth</code>
When <code>mem_mode</code> = 0 or 4, <code>eff_depth</code> = <code>ram_depth</code> + 1
When <code>mem_mode</code> = 1, 2, 5, or 6, <code>eff_depth</code> = <code>ram_depth</code> + 2
When <code>mem_mode</code> = 3 or 7, <code>eff_depth</code> = <code>ram_depth</code> + 3

Table 1-3 Parameter Description

Parameter	Values	Description
width	1 to 1024 Default: 8	Vector width of input data_s and output data_d
ram_depth	4 to 1024 Default: 8	Desired number of FIFO locations to be operated out of RAM not including the cache.
mem_mode	0 to 7 Default: 3	Memory control/datapath pipelining. Defines where and how many re-timing stages in RAM: <ul style="list-style-type: none"> <li>0: No pre or post retiming</li> <li>1: RAM data out (post) re-timing</li> <li>2: RAM read address (pre) re-timing</li> <li>3: RAM data out and read address re-timing</li> <li>4: RAM write interface (pre) re-timing</li> <li>5: RAM write interface and RAM data out re-timing</li> <li>6: RAM write interface and read address re-timing</li> <li>7: RAM data out, write interface and read address re-timing</li> </ul>
f_sync_type	0 to 4 Default: 2	Forward synchronization stages (direction from source to destination domains) <ul style="list-style-type: none"> <li>0: No synchronizing stages</li> <li>1: Two-stage synchronization with first stage negative-edge and second stage positive-edge capturing</li> <li>2: Two-stage synchronization with both stages positive-edge capturing</li> <li>3: Three-stage synchronization with all stages positive-edge capturing</li> <li>4: Four-stage synchronization with all stages positive-edge capturing</li> </ul>
r_sync_type	0 to 4 Default: 2	Return synchronization stages (direction from destination to source domains) <ul style="list-style-type: none"> <li>0: No synchronizing stages</li> <li>1: Two-stage synchronization with first stage negative-edge and second stage positive-edge capturing</li> <li>2: Two-stage synchronization with both stages positive-edge capturing</li> <li>3: Three-stage synchronization with all stages positive-edge capturing</li> <li>4: Four-stage synchronization with all stages positive-edge capturing</li> </ul>

**Table 1-3 Parameter Description (Continued)**

Parameter	Values	Description
clk_ratio	-7 to -1, 0, or 1 to 7 Default: 1	<p>Rounded quotient between <code>clk_s</code> and <code>clk_d</code> frequencies.</p> <p>NOTE: Ignored when <code>mem_mode</code> is 0 or 1, and should be set to the default value. See <a href="#">“When Is It Necessary to Determine clk_ratio”</a> on page 7.</p> <ul style="list-style-type: none"> <li>1 to 7: When <code>clk_d</code> rate faster than <code>clk_s</code> rate: <math>\text{round}(\text{clk\_d rate} / \text{clk\_s rate})</math></li> <li>-7 to -1: When <code>clk_d</code> rate slower than <code>clk_s</code> rate: <math>0 - \text{round}(\text{clk\_s rate} / \text{clk\_d rate})</math></li> <li>0: No restriction on clock <code>clk_s</code> and <code>clk_d</code> relationship (will incur a small performance degradation to retime synchronized pointers into each clock domain)</li> </ul>
rst_mode	0 or 1 Default: 0	<p>Control reset of RAM contents</p> <ul style="list-style-type: none"> <li>0: Include resets to clear RAM</li> <li>1: Do not include reset to clear RAM</li> </ul>
err_mode	0 or 1 Default: 0	<p>Error reporting</p> <ul style="list-style-type: none"> <li>0: Sticky error flag</li> <li>1: Dynamic error flag</li> </ul>
tst_mode	0 to 2 Default: 0	<p>Test mode</p> <ul style="list-style-type: none"> <li>0: No latch is inserted for scan testing</li> <li>1: Insert negative-edge capturing flip-flop on <code>data_s</code> input vector when test input is asserted</li> <li>2: Insert hold latch using active low latch.</li> </ul>
verif_en <sup>a</sup>	0 or 4 Default: 1	<p>Verification enable control</p> <ul style="list-style-type: none"> <li>0: No sampling errors inserted</li> <li>1: Sampling errors are randomly inserted with 0 or up to 1 destination clock cycle delays</li> <li>2: Sampling errors are randomly inserted with 0, 0.5, 1, or 1.5 destination clock cycle delays</li> <li>3: Sampling errors are randomly inserted with 0, 1, 2, or 3 destination clock cycle delays</li> <li>4: Sampling errors are randomly inserted with 0 or up to 0.5 destination clock cycle delays</li> </ul>
clr_dual_domain	0 or 1 Default: 1	<p>Activity of <code>clr_s</code> and/or <code>clr_d</code></p> <ul style="list-style-type: none"> <li>0: Either <code>clr_s</code> or <code>clr_d</code> can be activated, but the other must be tied low</li> <li>1: Both <code>clr_s</code> and <code>clr_d</code> can be activated</li> </ul>
arch_type <sup>b</sup>	0 or 1 Default: 0	<p>Pre-fetch cache architecture type</p> <ul style="list-style-type: none"> <li>0: Pipeline style (PL cache) - 'rtl' implementation</li> <li>1: Register File style (RF cache) - 'lpwr' implementation</li> </ul>

a. For more information about `verif_en`, see [“Simulation Methodology”](#) on page 28.

- b. For *arch\_type* equal to 1, the RF cache ('lpwr' implementation) is used when minPower is enabled and *mem\_mode* is not 0 or 4. If minPower is not enabled or *mem\_mode* is 0 or 4, the PL cache ('rtl' implementation) is always used regardless of the setting of *arch\_type*.

**Table 1-4 Synthesis Implementations**

Implementation Name	Function	License Feature Required
rtl	Synthesis model	DesignWare
lpwr <sup>a</sup>	Low Power synthesis model	<ul style="list-style-type: none"> <li>DesignWare (P-2019.03 and later)</li> <li>DesignWare-LP (before P-2019.03)</li> </ul>

- a. Requires that you enable minPower; for details, see [“Enabling minPower”](#) on page 43.  
When minPower is enabled, the lpwr implementation is always chosen during synthesis.

**Table 1-5 Simulation Models**

Model	Function
DW03.DW_FIFO_2C_DF_CFG_SIM	Design unit name for VHDL simulation
DW03.DW_FIFO_2C_DF_CFG_SIM_MS	Design unit name for VHDL simulation with mis-sampling enabled.
dw/dw03/src/DW_fifo_2c_df_sim.vhd	VHDL simulation model source code (modeling RTL) - no missampling
dw/sim_ver/DW_fifo_2c_df.v	Verilog simulation model source code

## When Is It Necessary to Determine *clk\_ratio*

The parameter *clk\_ratio* is only relevant when the parameter *mem\_mode* indicates there are retiming registers on the input (write port or read port or both) of the RAM being used for the FIFO. When *mem\_mode* is set to either 0 (no retiming registers in the RAM), or 1 (retiming registers only at the RAM data output port), the value of the parameter *clk\_ratio* doesn't matter and should use the default value. For designs without a fixed clock ratio, it is least restrictive to use a configuration with the parameter *mem\_mode* set to either 0 or 1, since the design will operate properly with any clock ratio -- with source faster than destination or destination faster than source at any ratio.

When the clock ratio is not fixed, as long as only one clock is always equal to or faster than the other, setting the *clk\_ratio* parameter to the highest clock ratio in the design would be sufficient for normal operation at all clock ratios.

The special case of setting *clk\_ratio* to 0 allows the design to operate properly regardless of the frequency relationship between *clk\_s* and *clk\_d* as well as for any RAM configuration (meaning for any value of the *mem\_mode* parameter). This is useful when a design needs to interface to a data stream with characteristics that are not known until a connection is made. However, if a design will always operate with a specific clock ratio, setting *clk\_ratio* to 0 could result in a design with more registers than necessary and lead to more latency than necessary.

## Detailed Description of Parameter *arch\_type*

The *arch\_type* parameter is available for selection of the pre-fetch cache structure that will reside in the data path after the RAM (in the underlying DW\_fifoctl\_2c\_df sub-component). This provides flexibility in choosing the best cache structure that yields the lowest power consumption based on system characteristics. For more about power aspects, see [“Pre-fetch Cache Architectures and Power Considerations”](#) on page 15.

If *arch\_type* is 0, the DW\_fifo\_2c\_df uses the pipeline (PL) cache structure which is the rtl implementation. When *arch\_type* is 1 and the *mem\_mode* setting is such that the pre-fetch cache depth is two or three, then a register file (RF) style of caching (lpwr implementation) is used provided a DesignWare-LP license is available.

When the RF Cache structure is desired and this component is configured accordingly, the 'lpwr' implementation is automatically selected when a DesignWare-LP license is available. Only a “set implementation” to the rtl implementation will override the selection of the lpwr implementation. If no DesignWare-LP license is available, but the component is configured to 'attempt' to use the RF Cache (for example, the lpwr implementation), the rtl implementation will be used instead which pertains the PL cache structure. In general, whenever the PL Cache is desired and the component is configured as such then a DesignWare-LP license is not consumed.

[Table 1-6](#) shows how the *arch\_type* setting along with the *mem\_mode* setting and license availability determines the implementation and, hence, the style of pre-fetch cache that is utilized.

**Table 1-6 Implementation Availability Based on *arch\_type* and *mem\_mode***

<i>arch_type</i>	<i>mem_mode</i>	Implementation available
0	x	rtl
1	4 or 5	rtl
1	1-3, 5-7	rtl, lpwr <sup>a</sup>

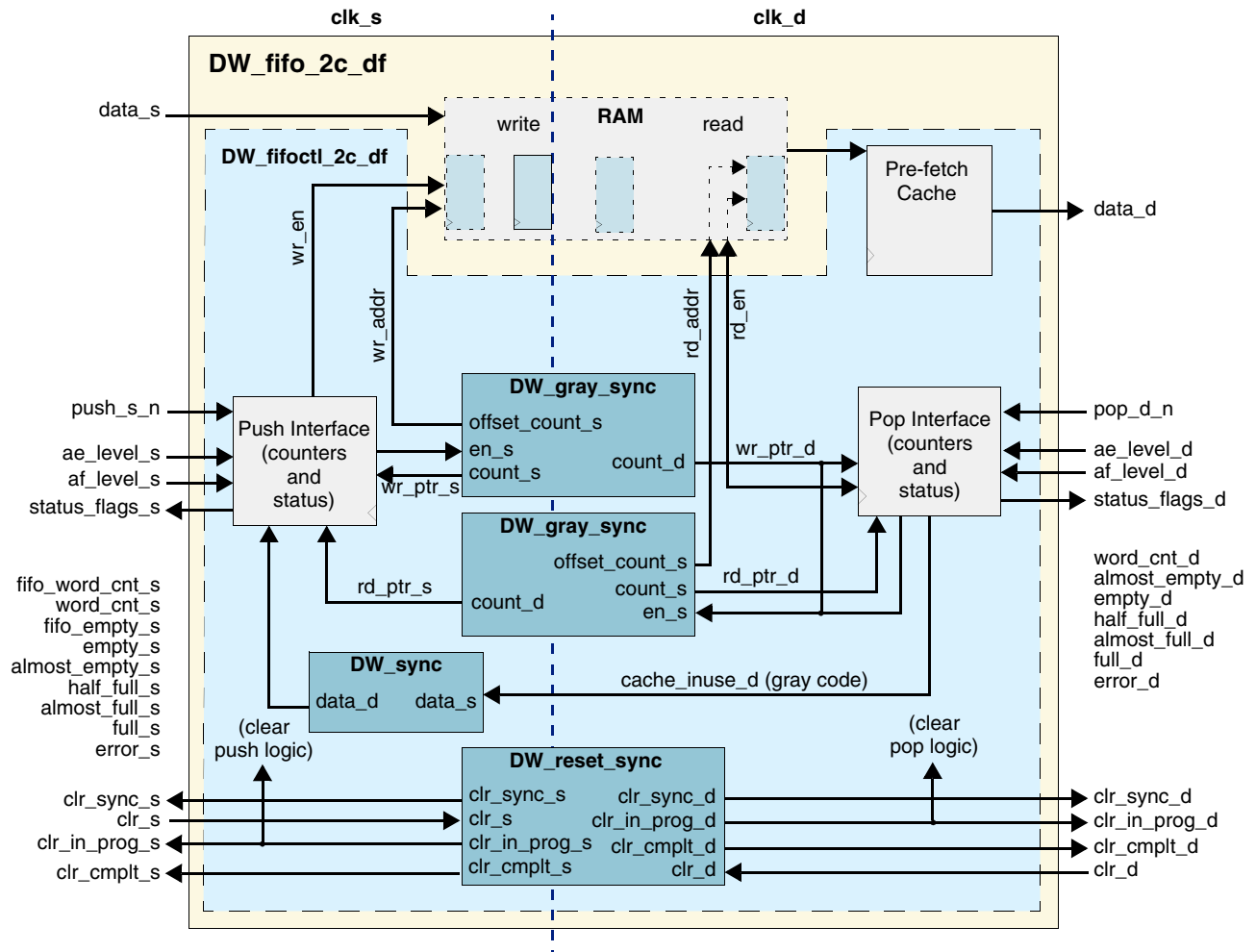
- a. For these settings of *arch\_type* and *mem\_mode*, the lpwr implementation is automatically selected instead of the rtl implementation unless overridden by 'set implementation'. To use the lpwr implementation, you must enable minPower; for details, see [“Enabling minPower”](#) on page 43. Also, for minPower licensing requirements, see [Table 1-4](#) on page 7.



## Block Diagram

Figure 1-1 show the block diagram of the DW\_fifo\_2c\_df.

Figure 1-1 DW\_fifo\_2c\_df Basic Block Diagram



## Reset Considerations

### System Resets (synchronous and asynchronous)

The system resets, `rst_s_n` and `init_s_n` for the source (push) domain and `rst_d_n` and `init_d_n` for the destination (pop) domain, work independently between the two domains. When the activation of these resets is not coordinated between the two domains at the system level, data corruption and false status reporting is possible.

The following are some guidelines on how the system resets between the two domains should be coordinated.

For system reset conditions, if the assertion of the resets occurs in one clock domain, the other domain must also assert its reset so that both domains are eventually in reset. That is, at some time in the system reset sequence, both domains must be in the active reset condition simultaneously. The length of the system reset signal(s) assertion must be a minimum of four clock cycles of the slowest clock between the two domains. Both clock domain system reset signals, when asserted, should overlap for a minimum of  $f\_sync\_type + 1$  or  $r\_sync\_type + 1$  cycles (which ever is larger) of the slowest of the two domains' clocks.

Besides satisfying simultaneous assertion of each domains system reset signals for a minimum number of cycles, the timing of the assertion between these signals needs some consideration. To prevent erroneous `clr_sync_s` and `clr_sync_d` pulses from occurring when system resets are asserted, it is recommended that:

- If the source domain system reset is asserted first, then the destination domain should assert its reset within  $f\_sync\_type + 1$  `clk_d` cycles from the time the assertion of the source domain reset occurred OR
- If the destination domain system reset is asserted first, then the source domain should assert its reset within  $r\_sync\_type + 1$  `clk_s` cycles from the time the assertion of the destination domain reset occurred.

If both domains can tolerate a false `clr_sync_s` or `clr_sync_d` (whichever the case) during system reset conditions, then this recommendation can be ignored as long as both clock domains eventually have overlapping active reset conditions.

There are no restrictions on when to release the reset condition on either side. However, to be completely safe, it is recommended, though not required, to release the source clock domain's reset last.

For examples of asynchronous and synchronous system reset assertion, see [Figure 1-14](#) on page 40 and [Figure 1-15](#) on page 41.

### rst\_mode Setting

The `rst_mode` parameter is available to allow the source domain system resets to clear the contents of the RAM (or not). When `rst_mode` is set to 0, the RAM contents are cleared on `rst_s_n` and/or `init_s_n` assertions. When `rst_mode` is set to 1, the assertion of `rst_s_n` and `init_s_n` have no affect on the RAM contents.

## Clearing FIFO (synchronous)

The DW\_fifo\_2c\_df contains one coordinated clearing signal from each domain called `clr_s` and `clr_d`. A minimum of a single clock cycle pulse on either one of these clearing signals initiates a synchronized clearing sequence to each domain for resetting of its sequential devices. This clearing sequence is orchestrated to ensure that the destination domain interface is completely cleared and ready for more data before the source domain is permitted to begin sending.

Figure 1-11 on page 37 and Figure 1-12 on page 38 show the clearing sequence for when `clr_s` and `clr_d`, respectively, are asserted. Additionally, Figure 1-13 on page 39 shows another `clr_s` initiated clearing sequence with a `clr_s` that is asserted for longer than a single `clk_s` cycle.

It is imperative for data transfer integrity to cease pushing any packets after asserted `clr_s` (and while waiting for a subsequent `clr_cmplt_s` pulse) and/or when observing an active `clr_in_prog_s`. From the destination domain, reading data packets after `clr_d` is asserted and/or observing an active `clr_in_prog_d` would result in corrupt data packet retrieval. Bottom-line, it is very important to halt pushing and popping during the clearing sequence and only start pushing new data after the `clr_cmplt_s` pulse is observed. Also, during the clearing sequence from when the `clr_s` is asserted to `clr_cmplt_s` assertion for the `clr_s` initiated case OR from the time `clr_d` is asserted to `clr_cmplt_s` assertion, it is important to realize that the values of all off status flags and word counts in both domains will not be reliable. Only after the completion of the coordinated clearing sequence are the status flags and word counts accurate. Figure 1-11 on page 37 (`clr_s` initiated clearing) shows an example of this.

There is no restriction on how often or how long `clr_s` and `clr_d` can be asserted. The clearing operation is maintained if in progress and subsequent `clr_s` and/or `clr_d` initiations are made. Once the final assertion of `clr_s` and/or `clr_d` is made, the sustained clearing sequence eventually comes to completion and all in-progress flags de-assert.

## Test

The synthesis parameter, `tst_mode`, controls the insertion of lock-up latches at the points where signals cross between the clock domains, `clk_s` and `clk_d`. Lock-up latches are used to ensure proper cross-domain operation during the capture phase of scan testing in devices with multiple clocks. When `tst_mode = 1`, lock-up latches are inserted during synthesis and are controlled by the input `test`.

With `tst_mode = 1`, the input, `test`, controls the bypass of the latches for normal operation where `test = 0` bypasses latches and `test = 1` includes latches. In order to assist DFT compiler in the use of the lock-up latches, use the “`set_test_hold 1 tst_mode`” command before using the `insert_scan` command.

When `tst_mode = 0` (which is its default value when not set in the design) no lock-up latches are inserted and the `test` input is not connected.



### Note

The insertion of lock-up latches requires the availability of an active low enable latch cell. If the target library does not have such a latch or if latches are not allowed (using `dont_use` commands for instance), synthesis of this module with `tst_mode = 1` will fail.

## Memory Depth Considerations and Setting *ram\_depth*

Depending on the desired FIFO depth of the design, the *ram\_depth* must be set accordingly.

Ultimately, the RAM must contain an even number of locations. Based on the desired number of FIFO locations, consider the following three cases in choosing the RAM size and *ram\_depth* setting.

**Case 1:** If an odd number of FIFO locations are required by the system (from the push interface), call that 'x', then the parameter *ram\_depth* should be set to 'x'. But, the RAM size should be 'x' + 1. The FIFO controller RAM addresses range from 0 to *ram\_depth*.

**Table 1-7** Desired Number of FIFO Locations is Odd

FIFO Locations Desired	RAM Size	RAM Address Range	<i>ram_depth</i> value
11	12	0 to 11	11
31	32	0 to 31	31

**Case 2:** If an even number of FIFO locations is required by the system (from the push interface) but that number is not an integer power of 2, call it 'y' locations, then the parameter *ram\_depth* should be set to 'y'. But, the size of the RAM should be 'y' + 2. The FIFO controller RAM addresses range from 0 to *ram\_depth* + 1.

**Table 1-8** Desired Number of FIFO Locations Is Even but Not a Power of 2

FIFO Locations Desired	RAM Size	RAM Address Range	<i>ram_depth</i> value
12	14	0 to 13	12
34	36	0 to 35	34

If an even number of FIFO locations is needed in the system (from the push interface) and it is an integer power of 2 (4, 8, 16, 32, ...), then set the *ram\_depth* to exactly the desired FIFO depth and the number of RAM locations accessed will also be the value of *ram\_depth*. The FIFO controller RAM addresses range from 0 to *ram\_depth* - 1.

**Table 1-9** Desired Number of FIFO Locations Is Even and a Power of 2

FIFO Locations Desired	RAM Size	RAM Address Range	<i>ram_depth</i> value
32	32	0 to 31	32
128	128	0 to 127	128

These restrictions are derived from the following facts:

- The memory depth must always be an even number to permit all transitions of the internal Gray coded pointers to be Gray (DW\_gray\_sync).
- For non-power of 2 depths, the memory size must be at least one greater than *ram\_depth* to allow the pointer arithmetic to unambiguously differentiate between the empty and full states.

## Writing to the Memory (Push)

A push is executed when:

- The `push_s_n` input is asserted (active low), and
- The `full_s` flag is inactive (low) at the rising edge of `clk_s`.

Asserting `push_s_n` when `full_s` is inactive causes the FIFO to prepare to write the value at the `data_s` input port to the internal RAM. On the next rising edge of `clk_s`, the `data_s` is written into the RAM and the internal write address pointer is advanced.

The `push_s_n` must be asserted early enough to propagate through the FIFO controller to the RAM before the ensuing clock.

## Write Errors

An error occurs if a push operation is attempted while the RAM (within the FIFO) is full. That is, the `error_s` output goes active if:

- The `push_s_n` input is asserted (low), and
- The `full_s` flag is active (high) on the rising edge of `clk_s`.

When a push error occurs, the data word that the application attempted to push onto the FIFO is lost and the internal write address does not advance. After a push error, even though a data word was lost at the time of the error, the FIFO remains in a valid full state and can continue to operate properly with respect to the data that was contained in the FIFO before the push error occurred.

## Destination Domain Caching (Pop Interface Pre-fetching)

The popping interface contains output buffering (pre-fetching cache) via the `DW_fifoctrl_2c_df` with the number of pipeline stages determined by the `mem_mode` parameter. When the cache is not fully populated with valid data and the RAM is detected as having valid entries, data is automatically pre-fetched into the cache to provide immediate data availability at pop requests no matter which mode of read port the RAM is using (asynchronous versus 1-deep synchronous versus two-deep synchronous). An extra latency applies not only to when the first data word arrives at the pop interface but also to when FIFO fullness information is delivered to the `word_cnt_d` and pop interface status flag ports.

At a minimum, there will always be at least one buffering stage in the cache which is seen at the pop interface. However, only the first word of a burst of words incurs a one `clk_d` cycle latency before being read from the RAM and presented to the pop interface. Below is a list identifying the number of pre-fetching stages of the cache used based on the `mem_mode` parameter value.

**Table 1-10 Pop Interface Cache Sizes**

<i>mem_mode</i> values	Number of caching stages
0 or 4	1
1, 2, 5, or 6	2
3 or 7	3

## Reading from the FIFO (Pop)

The cache of the DW\_fifoctl\_2c\_df is the data interface of the FIFO and it is made up of pipelined data words based on the *mem\_mode* parameter as described in Table 1-10 on page 13. When the head of the cache (the *data\_d* output port) contains a valid entry the FIFO is considered not empty (the *empty\_d* flag is not asserted), a legal pop of the FIFO is allowed (asserting *pop\_d\_n*). When *empty\_d* is not asserted the *data\_d* contents is the next valid word from the FIFO. The assertion of *pop\_d\_n* causes the cache pipeline to shift valid data on the next rising edge of *clk\_d*. If active RAM data out internally is available, the RAM data output is loaded into the closest vacated stage to the head of the cache. For example, if only the head stage of the cache contains valid data and RAM data output is valid and *pop\_d\_n* is asserted, then on the next rising edge of *clk\_d* the RAM data output is loaded to the head of the cache. This event would keep *empty\_d* deasserted and allow for another pop on the next rising edge of *clk\_d*.

However, if only the head of the cache contains valid data and RAM data output is not valid and *pop\_d\_n* is asserted, then on the next rising edge of *clk\_d* the data value at the head of the cache (*data\_d*) is held BUT the *empty\_d* flag gets asserted. Thus, assertion of the *empty\_d* flag declares the contents at *data\_d* irrelevant.

It is worth noting that due to the pipelining nature of the read data path, the head of the cache may not contain relevant data, hence the FIFO is declared empty, but data from previous read operations could be in transit and making their way to the head of the cache. So, empty only means that the head of the cache does not contain relevant data, but there could still be relevant data being actively processed through the read data path.

## Read Errors

An error occurs if a pop operation is attempted while the FIFO is empty (as perceived by the pop interface). That is, the *error\_d* output goes active if:

- The *pop\_d\_n* input is active (low), and
- The *empty\_d* flag is active (high) on the rising edge of *clk\_d*.

When a pop error occurs, the internal read address does not advance. After a pop error the FIFO is still in a valid empty state and can continue to operate properly.

## Error Outputs and Flag Status

The error outputs and flags are initialized as follows:

- *fifo\_empty\_s*, *empty\_s*, *almost\_empty\_s*, *empty\_d*, and *almost\_empty\_d* are initialized to 1 (high)
- All other flags and the error outputs are initialized to 0 (low)

## Pre-fetch Cache Architectures and Power Considerations

As mentioned earlier, there are two pre-fetch cache architectures (that reside in the underlying DW\_fifoctrl\_2c\_df component) which are parameter selectable that allows for power optimization: pipelined (PL) and register file (RF) types.

The PL caching style (rtl implementation) is effectively a shift register of one, two, or three stages. Active switching through each stage occurs during shifting initiated by pop requests with pending valid data in either the RAM or cache stages behind the head location. In cases with a wide data bus and cache configurations of two or three deep, this could represent the majority of the register switching power consumption within the component.

As an alternative architecture for cache depths of two or three, the pre-fetch cache is organized as a RF structure ('lpwr' implementation). For the RF cache structure, the shifting between pipelined cache entries is eliminated and replaced with write and read pointer manipulation to access cache elements; a mini-FIFO of sorts.

The two caching architectures are provided to give the designer flexibility in selecting the caching architecture that will yield the lowest total power consumption.

Knowing which pre-fetch cache architecture to choose is highly dependent on factors such as technology, clock rate, data width, pre-fetch cache depth, ram depth, and data flow characteristics through the FIFO.

With all variables being equal, the advantage that either cache architecture provides, in terms of optimal power dissipation, is mostly based on the type of data flow behavior through the DW\_fifo\_2c\_df.

Generally, there is no specific rule of thumb in selecting which cache architecture will render the least power dissipation. This may require the design process to include some experimentation in using both cache styles to characterize behavior based on the system parameters.

Keep in mind, criticality in choosing which pre-fetch cache architecture is only meaningful in a system when the parameter *mem\_mode* is not 0 or 4. That is, when the cache depth is two or three, the selection of the cache architecture is relevant. When *mem\_mode* is either 0 or 4, the pre-fetch cache depth is one and defaults to the 'rtl' implementation.

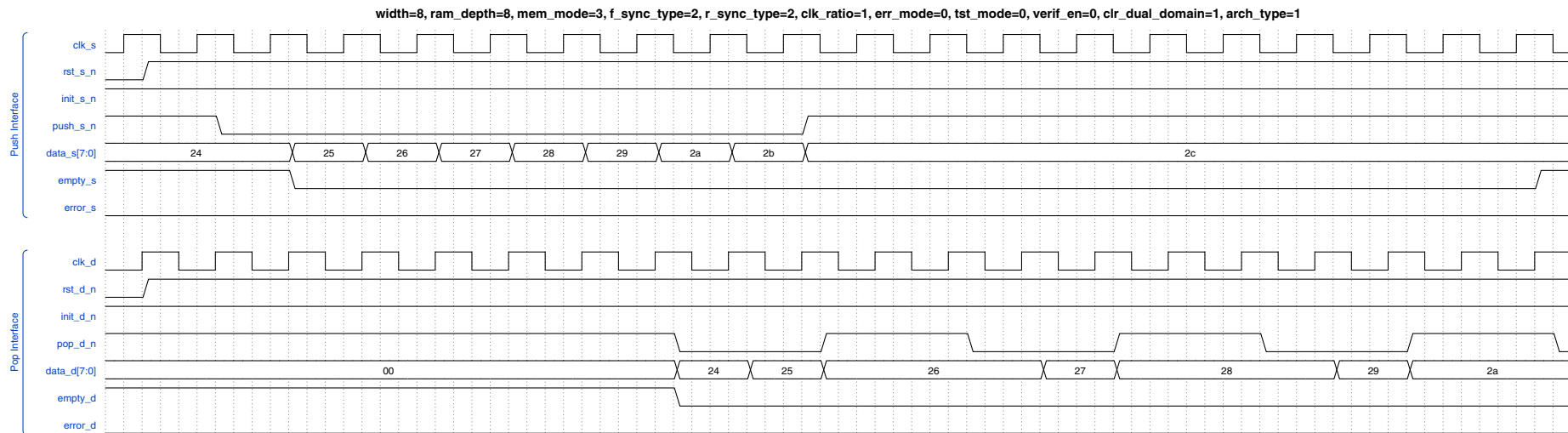
However, as a starting point, there are two data flow behaviors and the cache architecture that will most likely yield a better power result over the other. Of course, this is assuming that this data flow is a significant power consumption factor through the DW\_fifoctrl\_2c\_df.

The key factor that determines which cache architecture will provide the least power consumption over the other hinges on the behavior of the pop requests (pop\_d\_n). If pop requests are issued in short bursts of three or less, the RF cache ('lpwr' implementation) will most likely yield the least power consumption versus the PL cache architecture ('rtl' implementation). If however, pop requests that occur in longer contiguous bursts favor the PL cache architecture. The following two cases show the two extremes in data flow behavior.



## CASE 1: Push Packets and Pop Alternately with Two Cycles Active then Two Cycles Inactive

Figure 1-2 Data Flow of Popping Small Bursts



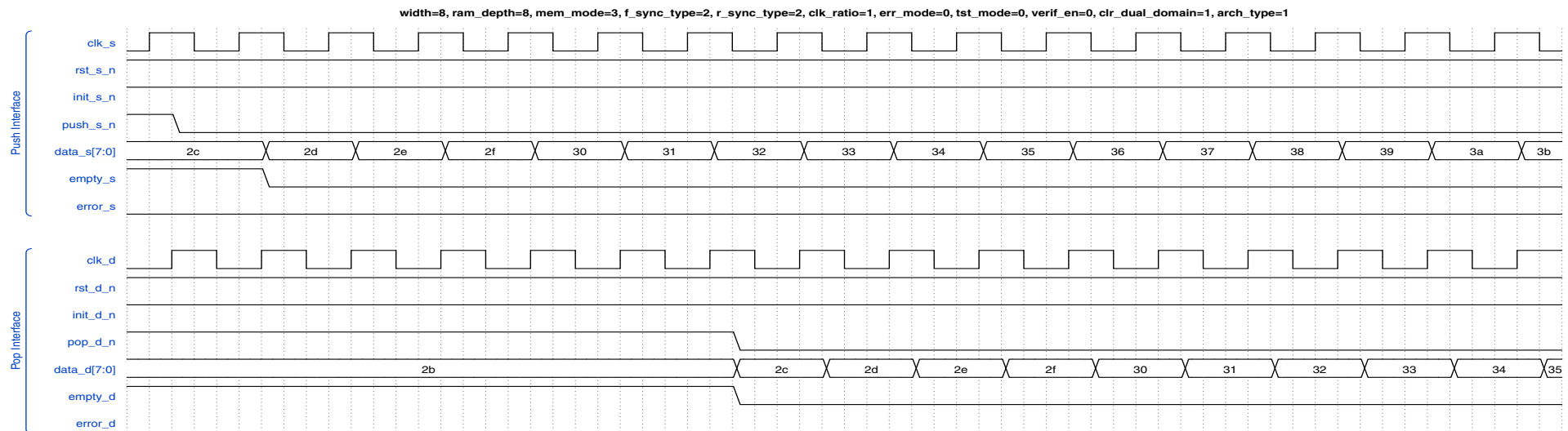
The data flow behavior for [Figure 1-2](#) shows a packet of length *depth* (8 in this case) with words pushed contiguously before pushing halts. The pop activity begins as the FIFO is approximately half full and follows a progression of two cycles active and two cycles idle. This particular data flow, with a cache depth of three, is the best-case behavior that favors the selection of the Register File (RF) cache architecture (*arch\_type* of 1 and *mem\_mode* of 3 or 7). Similarly, the pop request being active every other cycle for a cache depth of two (*arch\_type* of 1 and *mem\_mode* of 1, 2, 5, or 6) would be the best-case data flow behavior geared to selecting the RF cache style. The disparity in power savings increases with larger *width* parameter values because a larger data path portion begins to dwarf the other portions of the component and, thus, the data flow is a more prominent contributor to the overall dynamic power. That is, with larger *width* parameter values, the data flow through the component in the PL cache produces larger dynamic power numbers than the RF cache architecture. Thus, the advantage of using the RF cache style is greater. Conversely, as RAM depth increases along with widening data widths, the RAM power becomes a larger portion of the overall power consumption and will tend to proportionally reduce the influence that power savings gained from the use of RF cache implementation.

For example, taking into account the above pop\_d\_n behavior (popping small bursts), a FIFO with *data\_width* of 32 and *ram\_depth* of 8 (compared to a *data\_width* of 32 and a *ram\_depth* of 64) most likely results in a larger percentage improvement of the RF cache over the PL cache for the *ram\_depth* of 8 because the RAM itself is much smaller and its total power consumption much less than the RAM with *ram\_depth* of 64. Thus, the absolute power consumption difference of the RF cache compared to the PL cache is a bigger proportion of the overall power consumption of the component leading to a bigger percentage benefit. That is, increasing the size of RAM is inversely proportional to the percentage of power savings gained from using the RF cache over the PL cache. Again, the benefits vary based on system behavior, configuration, technology, and so on.



## CASE 2: Popping in Long Contiguous Bursts

Figure 1-3 Data Flow of Popping Continuously



When long bursts of contiguous pop requests are issued (Figure 1-3), this produces a type of data flow more conducive to using the PL cache architecture. The power benefits of using the PL cache come from the fact that the front of the cache is continuously being written to and read from. From the PL cache perspective one stage of the cache is being used at a time and effectively the other stages of the cache are unused during this time. So, the dynamic power of shifting through many stages of the cache does not occur. In the RF cache however, data is being written to cache, just like in the PL cache case, but the write and read addressing logic is updating every cycle. This activity of the write and read addressing logic is the extra power consumption that the RF cache architecture has that the PL cache does not. Thus, the PL cache architecture would be optimal, in general, for this type of data flow.

Note that in CASE 1 (Figure 1-2), the write and read address logic is always active as well. But the difference compared to CASE 2 is that the data flow through the cache caused by the bursting pop requests is such that the cache is full, almost full, or becoming full. Thus, all the cache locations are shifting data in or out on every cycle. Therefore, power consumption of the cache is predominantly the shifting of data (inherent to the PL cache) and not the write and read address logic. Thus, selecting the RF cache, in general, provides the best power consumption results in the CASE 1 scenario. Again, the differences in favor of the RF cache over the PL cache in this data flow behavior increase as data widths increase.

## Synchronization Between Clock Domains

Each interface (source domain (push) and destination domain (pop)) operates synchronous to its own clock: `clk_s` and `clk_d`, respectively. Each interface is independent, containing its own state machine and flag logic. The pop interface has the primary read address counter and a synchronized copy of the write address counter. The push interface has the primary write address counter and a synchronized copy of the read address counter. The two clocks may be asynchronous with respect to each other. The FIFO controller performs inter-clock synchronization in order for each interface to monitor the actions of the other. This enables the number of words in the FIFO at any given point in time to be determined independently by the two interfaces.

The only information that is synchronized across clock domain boundaries is the read or write address generated by the opposite interface. If an address is transitioning while being sampled by the opposite interface (for example, `wr_addr_s` sampled by `clk_d`), sampling uncertainty can occur. By Gray coding the address values that are synchronized across clock domains, this sampling uncertainty is limited to a single bit. Single bit sampling uncertainty results in only one of two possible Gray coded addresses being sampled: the previous address or the new address. The uncertainty in the bit that is changing near a sampling clock edge directly corresponds to an uncertainty in whether the new value will be captured by the sampling clock edge or whether the previous value will be captured (and the new value may be captured by a subsequent sampling clock edge). Thus, there are no errors in sampling Gray coded pointers, just a matter of whether a change of pointer value occurs in time to be captured by a given sampling clock edge or whether it must wait for the next sampling clock edge to be registered. To do this transporting of Gray code addressing, the DesignWare component `DW_gray_sync` is instantiated for both directions.

### `f_sync_type` and `r_sync_type`

The `f_sync_type` and `r_sync_type` parameters determine the number of register stages used to synchronize the internal Gray code read pointer to `clk_s` (represented by `r_sync_type`) and internal Gray code write pointer to `clk_d` (represented by `f_sync_type`). For legal values of these parameters, see [Table 1-3](#) on page 5.

Two-stage synchronization is desirable when using relatively high clock rates. It allows an entire clock period for meta-stable events to settle at the first stage before being cleanly clocked into the second stage of the synchronizer. Two-stage synchronization increases the latency between the two interfaces, resulting in flags that are less up-to-date with respect to the true state of the FIFO.

Three-stage synchronization is also desirable when using very high clock rates. It allows an entire clock period for meta-stable events to settle at the first stage before being clocked into the second stage of the synchronizer. Then, in the unlikely event that a meta-stable event propagates into the second stage, the output of the second stage is allowed to settle for another entire clock period before being clocked into the third stage. Three-stage synchronization increases the latency between the two interfaces, resulting in flags that are less up-to-date with respect to the true state of the FIFO.

Four-stage synchronization is desirable in extreme differences in clock rates between the two domains.

### Empty to Not Empty Transitional Operation

When the FIFO is empty, `empty_s` and `empty_d` are active (high). During the first push (`push_s_n` active (low)), the rising edge of `clk_s` writes the first word into the FIFO. Initially, the `empty_s` flag is driven low.

The `empty_d` flag does not go low until 1 to 4 cycles (of `clk_d`) after the new internal Gray code write pointer has been synchronized to `clk_d`. This could be as long as two to eight cycles (depending on the

values of the *f\_sync\_type* and *mem\_mode* parameters). Refer to the timing diagrams for more information. The system design should allow for this latency in the depth budgeting of the FIFO design.

The *empty\_d* flag is based on the validity of the data sitting at the head of the cache. It does not represent a count value of valid data entries in the RAM and cache pipeline. To identify precisely the number of valid data entries in the FIFO, refer to the *word\_cnt\_d* output port that gives the updated FIFO word count from the pop interface perspective.

## RAM Not Empty to RAM Empty Transitional Operation

When the RAM module is almost empty, the *empty\_s* is inactive (low), *almost\_empty\_d* and the *empty\_d* could be either state depending on the cache state. When the last word of the RAM is retrieved to populate an available spot in the cache, the next rising edge of *clk\_d* causes the *empty\_d* flag to be driven low.

The *empty\_s* flag is not asserted (high) until one cycle (of *clk\_s*) after the new internal Gray code read pointer has been synchronized to *clk\_s*. This could be as long as two to six cycles (depending on the value of the *r\_sync\_type* parameter) from the time the read pointer changed in the pop domain. Refer to the timing diagrams for more information.

You should be aware of this latency when designing the system data flow protocol.

## Note About Full Status

The concept of full with respect to each domain is different because of the cache that resides in the destination domain (pop interface). In the source domain (push interface), the concept of full is with respect to the RAM module contents based on *full\_s*. In the destination domain, the concept full is with respect to the RAM module and cache contents based on *full\_d*. In describing the dynamics of full in the following two sections, the starting point is always in the perspective of the destination domain.

## Full to Not Full Transitional Operation

When the FIFO is full (RAM and cache full), both *full\_s* and *full\_d* are active (high). During the first pop (*pop\_d\_n* active low), the rising edge of *clk\_d* reads the first word out of the FIFO. The *full\_d* flag is driven low.

The *full\_s* flag does not go low until one cycle (of *clk\_s*) after the new internal Gray code read pointer has been synchronized to *clk\_s*. This could be as long as two to six cycles (depending on the value of the *r\_sync\_type* parameter) from the time the read pointer in the destination is updated. Refer to the timing diagrams for more information.

You should be aware of this latency when designing the system data flow protocol.

## Not Full to Full Transitional Operation

When the RAM is almost full (with respect to the source domain) both *full\_s* and *full\_d* are inactive (low) and *almost\_full\_s* is active (high). During the final push (*push\_s\_n* active (low)) assuming no pops and the cache is fully populated, the rising edge of *clk\_s* writes the last word into the RAM. The *full\_s* flag is driven high.

The *full\_d* flag is not asserted (high) until one cycle (of *clk\_d*) after the new internal Gray code write pointer has been synchronized to *clk\_d* (only after *full\_s* is asserted and the cache is full). This could be as

long as two to six cycles (depending on the value of the *f\_sync\_type* parameter) from when the write pointer in the source domain got updated. Refer to the timing diagrams for more information.

You should allow for this latency in the depth budgeting of the FIFO design.

## Errors

### err\_mode

The *err\_mode* parameter determines whether the *error\_s* and *error\_d* outputs remain active until reset (persistent) or for only the clock cycle in which the error is detected (dynamic).

When the *err\_mode* parameter is set to 0 at design time, persistent error flags are generated. When the *err\_mode* parameter is set to 1 at design time, dynamic error flags are generated.

### error\_s Output

The *error\_s* output signal indicates that a push request was seen while the *full\_s* output was active (high) (an overrun error). When an overrun condition occurs, the internal write address pointer does not advance, and the internal RAM write enable is not activated.

Therefore, a push request that would overrun the FIFO is, in effect, rejected, and an error is generated. This guarantees that no data already in the FIFO is destroyed (overwritten).

Other than the loss of the data accompanying the rejected push request, FIFO operation can continue without reset.

### error\_d Output

The *error\_d* output signal indicates that a pop request was seen while the *empty\_d* output signal was active (high) (an underrun error). When an underrun condition occurs, the internal read address pointer does not decrement, as there is no data in the FIFO to retrieve.

The FIFO timing is such that the logic controlling the *pop\_d\_n* input would not see the error until nonexistent data had already been registered by the receiving logic. This is easily avoided if this logic can pay close attention to the *empty\_d* output and thus avoid an underrun completely.

## Controller Status Flag Outputs

The two halves of the FIFO controller each have their own set of status flags indicating their separate view of the state of the FIFO. It is important to note that both the push interface and the pop interface perceives the state of fullness of the FIFO independently based on information from the opposing interface that is delayed up to three clock cycles for proper synchronization between clock domains. Also, due to the cache present in the destination domain (pop interface) fullness will be based on RAM and cache contents whereas the source domain (push interface) only considers the RAM contents for its fullness. The same is true for the state of emptiness with one exception regarding *empty\_d*.

The push interface status flags respond immediately to changes in state caused by push operations but there is delay between pop operations and corresponding changes of state of the push status flags. This delay is due to the latency introduced by the registers used to synchronize the internal Gray coded read pointer and prefetch cache count to *clk\_s*. The pop interface status flags respond immediately to changes in state caused by pop operations but there is delay between push operations and corresponding changes of state of

the pop status flags. This delay is due to the latency introduced by the registers used to synchronize the internal Gray coded write pointer to `clk_d`.

Most status flags have a property which is potentially useful to the designed operation of the FIFO controller. These properties are described in the following explanations of the flag behaviors.

## empty\_s

The `empty_s` output, active high, is synchronous to the `clk_s` input. `empty_s` indicates to the push interface that the RAM module is empty. During the first push, the rising edge of `clk_s` causes the first word to be written into the RAM, and `empty_s` is driven low.

The action of the last word being popped from a nearly empty RAM module is controlled by the pop interface. Thus, the `empty_s` output is asserted only after the new internal Gray code read pointer (from the pop interface) is synchronized to `clk_s` and processed by the status flag logic.

### Property of empty\_s

If `empty_s` is active (high) then the RAM module is truly empty. This property does not apply to `empty_d`.



#### Attention

When using push status outputs to make decisions on writing into the FIFO, use `empty_s` and `word_cnt_s` instead `fifo_empty_s` and `fifo_word_cnt_s`. The `empty_s` and `word_cnt_s` signals provide accurate information about how much space is available for writing into the RAM of the FIFO. For detailed information about `fifo_empty_s` and `fifo_word_cnt_s`, see [“Behavior of fifo\\_empty\\_s and fifo\\_word\\_cnt\\_s”](#) on page 22.

## almost\_empty\_s

The `almost_empty_s` output, active high, is synchronous to the `clk_s` input. The `almost_empty_s` output indicates to the push interface that the RAM module is almost empty when there are no more than `ae_level_s` (input port) words currently in the RAM module to be popped as perceived at the push interface.

The `ae_level_s` input port defines the almost empty threshold with respect to the RAM module of the push interface independent of that of the pop interface. The `almost_empty_s` output is useful when it is desirable to push data into the RAM module in bursts (without allowing the RAM module to become empty).

### Property of almost\_empty\_s

If `almost_empty_s` is active (high) then the RAM module has at least  $(ram\_depth - ae\_level\_s)$  available locations. Therefore such status indicates that the push interface can safely and unconditionally push  $(ram\_depth - ae\_level\_s)$  words into the RAM module. This property guarantees that such a “blind push” operation will not overrun the RAM module.

## half\_full\_s

The `half_full_s` output, active high, is synchronous to the `clk_s` input, and indicates to the push interface that the RAM module has at least half of its memory locations occupied as perceived by the push interface.

## Property of half\_full\_s

If half\_full\_s is inactive (low) then the RAM module has at least half of its locations available. Thus such status indicates that the push interface can safely and unconditionally push  $(\text{INT}(\text{ram\_depth}/2)+1)$  words into the RAM module. This property guarantees that such a “blind push” operation will not overrun the RAM module.

## almost\_full\_s

The almost\_full\_s output, active high, is synchronous to the clk\_s input. The almost\_full\_s output indicates to the push interface that the RAM module is almost full when there are no more than af\_level\_s empty locations in the RAM module as perceived by the push interface.

The af\_level\_s input port defines the almost full threshold with respect to the RAM module of the push interface independent of the pop interface. The almost\_full\_s output is useful when more than one cycle of advance warning is needed to stop the flow of data into the RAM module before it becomes full (to avoid a RAM module overrun).

## Property of almost\_full\_s

If almost\_full\_s is inactive (low) then the RAM module has at least (af\_level\_s+1) available locations. Thus such status indicates that the push interface can safely and unconditionally push (af\_level\_s+1) words into the RAM module. This property guarantees that such a “blind push” operation will not overrun the RAM module.

## full\_s

The full\_s output, active high, is synchronous to the clk\_s input. The full\_s output indicates to the push interface that the RAM module is full. During the final push, the rising edge of clk\_s causes the last word to be pushed, and full\_s is asserted.

The action of the first word being popped from a full RAM module is controlled by the pop interface. Thus, the full\_s output goes low only after the new internal Gray code read pointer from the pop interface is synchronized to clk\_s and processed by the status flag state logic.

## Behavior of fifo\_empty\_s and fifo\_word\_cnt\_s

The fifo\_empty\_s and fifo\_word\_cnt\_s status outputs are meant to indicate the *general* state of the FIFO. For example, when the FIFO is nearing empty, there can be a one clk\_s cycle window when the fifo\_empty\_s indicates “empty” when there is still one valid word stored in the FIFO. Also, the fifo\_word\_cnt\_s output value can be off by a  $\pm 1$  words for one clk\_s cycle before achieving steady state. That is, when at the “close to empty state”, the fifo\_word\_cnt\_s could report a '0' or '2' when there is actually '1' valid word in the FIFO. The reason behind this stems from there being two pieces of information from the pop domain that are synchronized and merged into the push domain, the “pop read pointer” and “pop pre-fetch cache count”. In the block diagram in [Figure 1-1](#) on page 9, these signals are “rd\_ptr\_d” and “cache\_inuse\_d”, respectively.

The “rd\_ptr\_d” and “cache\_inuse\_d” signals are then sent to the push domain (clk\_s) in parallel for synchronization. Although, together, they represent the correct count (one word in the FIFO) in the pop domain, their values may incur sampling issues at the push domain synchronizers due to logic delay and meta-stability, and the sampling issues can cause momentary instability in the push domain. This



momentary instability can result in inaccurate values on `fifo_word_cnt_s` and `fifo_empty_s` for one `clk_s` cycle. Steady state will occur, provided pushing and popping activities are suspended long enough for all the synchronization to stabilize.

Below are two sets of waveforms that show the possible inaccurate behavior of `fifo_word_cnt_s` and `fifo_empty_s` for a single word push and with no popping.

The following configuration values were used for [Figure 1-4](#) on page 24 and [Figure 1-5](#) on page 25:

*width* = 32

*ram\_depth* = 16

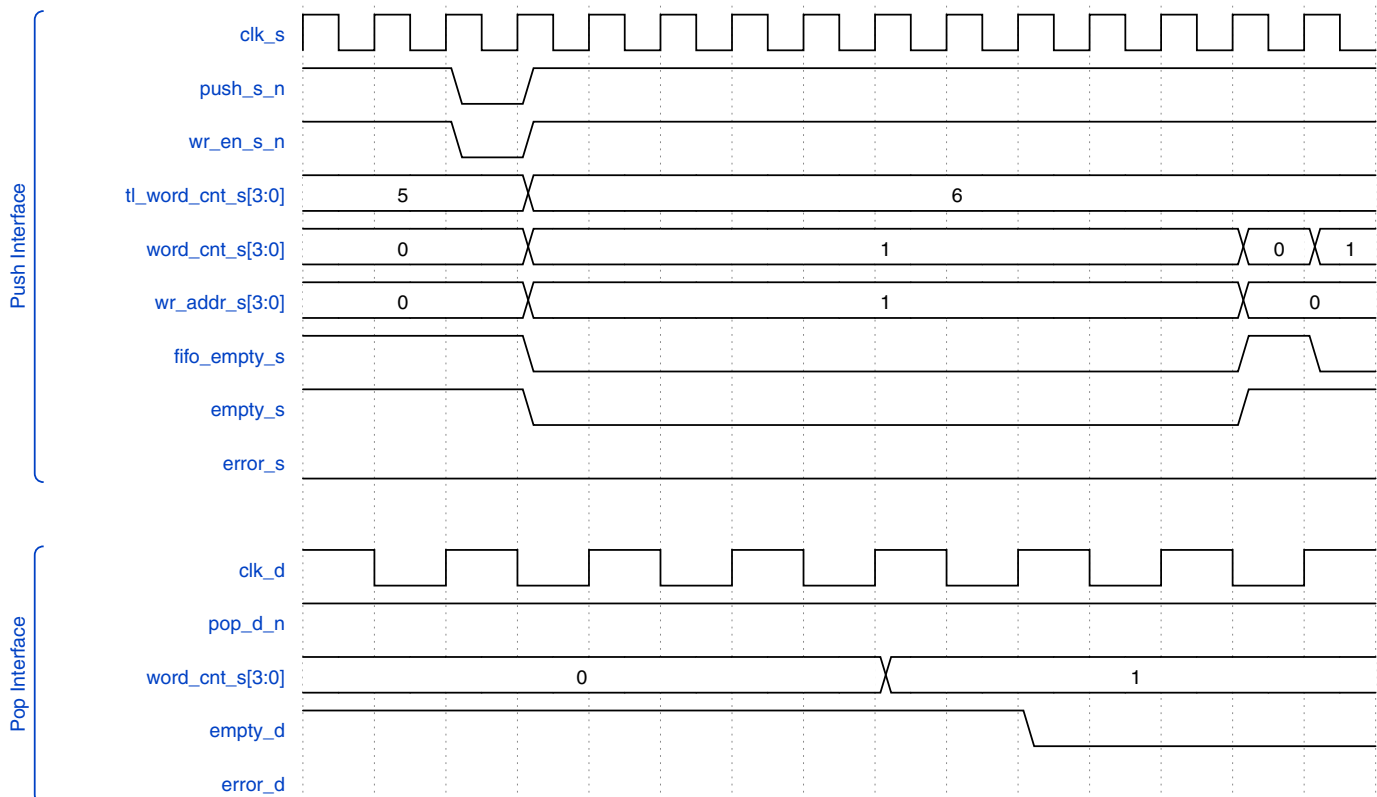
*f\_sync\_type* = 2

*r\_sync\_type* = 2

*mem\_mode* = 0

In Figure 1-4, the waveforms show a push request being made with `push_s_n` going to '0' for one `clk_s` cycle. Notice that no pop request is made throughout (`pop_n_d` stays at '1'). So, effectively, the FIFO has one valid word in it. As the pre-fetch to cache occurs (reading from the RAM), the read pointer and cache count from the pop domain are synchronized back to the push domain. In this case, `fifo_word_cnt_s` goes from '1' to '0' and then back to '1'. The transition to '0' is the instability, and is due to a synchronization sampling error causing a disparity between the read pointer and cache count in the push domain. This behavior is expected as `fifo_word_cnt_s` stabilizes at '1' after one `clk_s` cycle. Notice also that `fifo_empty_s` goes to '1' for a `clk_s` cycle coinciding with `fifo_word_cnt_s` being '0'.

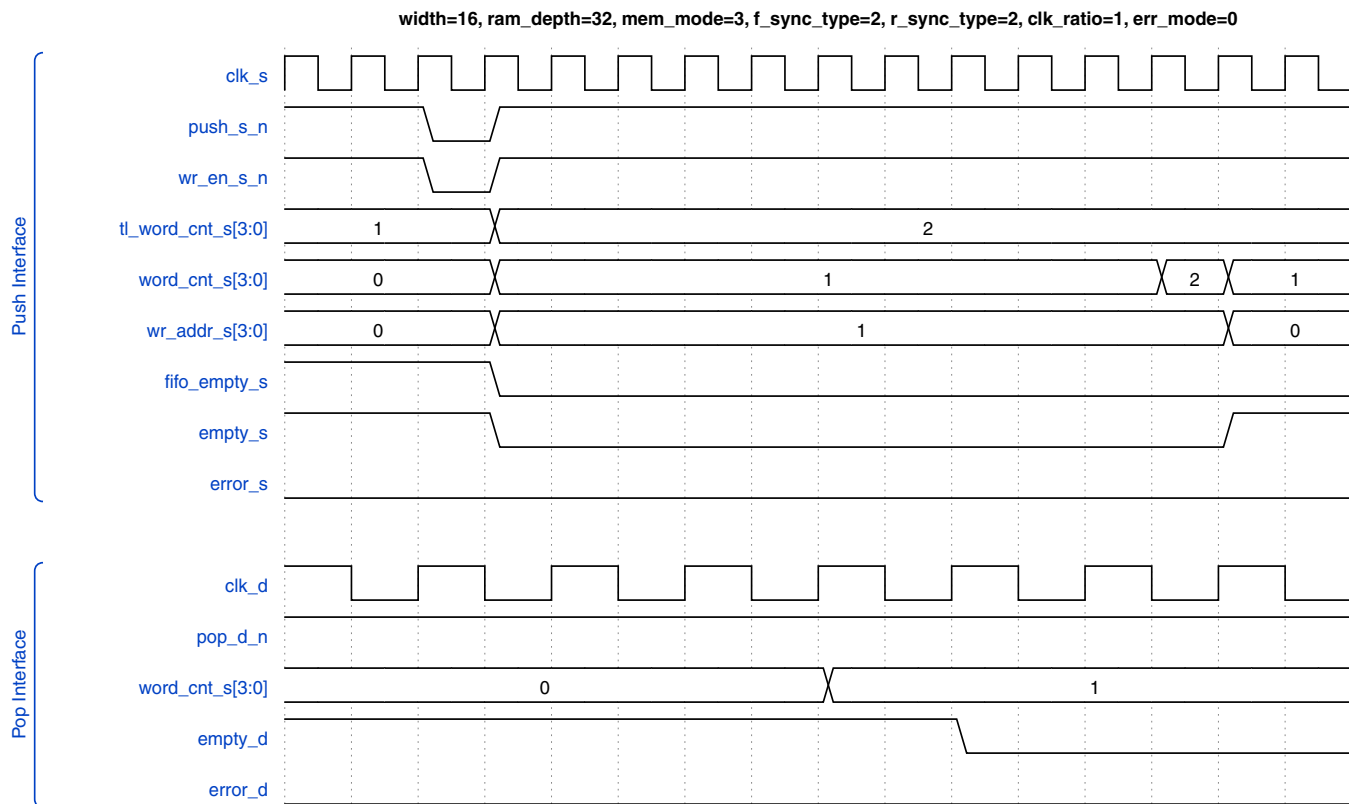
**Figure 1-4 Push Request Showing Instability On `fifo_word_cnt_s` (Example 1)**





In Figure 1-5, the waveforms show a push request being made with `push_s_n` going to '0' for one `clk_s` cycle. Notice that no pop request is made throughout (`pop_n_d` stays at '1'). So, effectively, the FIFO has one valid word in it. As the pre-fetch to cache occurs (reading from the RAM), the read pointer and cache count from the pop domain are synchronized back to the push domain. In this case, `fifo_word_cnt_s` goes from '1' to '2' and then back to '1'. The transition to '2' is the instability, and is due to a synchronization sampling error causing a disparity between the read pointer and cache count in the push domain. This behavior is expected as `fifo_word_cnt_s` stabilizes at '1' after one `clk_s` cycle.

**Figure 1-5 Push Request Showing Instability On `fifo_word_cnt_s` (Example 2)**



## Blind Pushing

*Blind pushing* is the operation of performing consecutive-cycle pushing without the risk of RAM overruns (overflows). The number of consecutive pushing cycles is predicated on the setting of level thresholds (`af_level_s` and/or `ae_level_s`) and the initiation of pushing is based on the state of the source domain status flags (`almost_full_s` and/or `almost_empty_s`, respectively, `half_full_s`, or `full_s`). In general, any pushing operation must rely on one or more of the provided source domain status flags (including `empty_s`) to know when pushing can begin and to prevent overrun. A common practice for implementing blind pushing operations is to use the `af_level_s` input value coupled with the `almost_full_s` status flag. For example, if the `af_level_s` is set to 2 and the source domain interface `almost_full_s` status flag is 0, then it would be safe to begin pushing (`push_s_n` to logic 0) consecutive operations of duration 3 (`af_level_s` + 1) without overrunning the RAM after which the push request MUST be released (for example, `push_s_n` must go to logic 1).

## empty\_d

The `empty_d` output, active high, is synchronous to the `clk_d` input. The `empty_d` flag indicates to the pop interface that the head of the cache does not contain relevant data. It does not necessarily mean that the RAM module or other stages of the cache contain invalid data. The action of the last word being popped from a nearly empty FIFO is controlled by the pop interface. Thus, the `empty_d` output is asserted at the rising edge of `clk_d` that causes the last word to be popped from the FIFO.

The last word in this context refers to when the RAM module is empty and the only relevant data word is sitting at the head of the cache pipeline.

The action of pushing the first word into an empty FIFO is controlled by the push interface. That means `empty_d` goes low only after the new internal Gray code write pointer from the push interface is synchronized to `clk_d`, data is read from the RAM module, and then placed into the cache.

## Property of empty\_d

If `empty_d` is active (high), then the head of the pre-fetch cache does not contain a valid data word.

## almost\_empty\_d

The `almost_empty_d` output, active high, is synchronous to the `clk_d` input. `almost_empty_d` indicates to the pop interface that the FIFO is almost empty when there are no more than `ae_level_d` (input port) words currently in the FIFO to be popped.

The `ae_level_d` input port defines the almost empty threshold with respect to the entire FIFO of the pop interface independent of the push interface. The `almost_empty_d` output is useful when more than one cycle of advance warning is needed to stop the popping of data from the FIFO before it becomes empty (to avoid a FIFO underrun).

## Property of almost\_empty\_d

If `almost_empty_d` is inactive (low) then there are at least (`ae_level_d` + 1) words in the FIFO. Therefore such status indicates that the pop interface can safely and unconditionally pop (`ae_level_d` + 1) words out of the FIFO. This property guarantees that such a “blind pop” operation will not underrun the FIFO.

## half\_full\_d

The `half_full_d` output, active high, is synchronous to the `clk_d` input. The `half_full_d` output indicates to the pop interface that the FIFO has at least half of its memory locations occupied.

### Property of half\_full\_d

If `half_full_d` is active (high) then at least half of the words in the FIFO are occupied. Therefore such status indicates that the pop interface can safely and unconditionally pop  $\text{INT}((\text{eff\_depth} + 1)/2)$  words out of the FIFO, where `eff_depth` is defined in Table 1-2 on page 4. This property guarantees that such a “blind pop” operation will not underrun the FIFO.

## almost\_full\_d

The `almost_full_d` output, active high, is synchronous to the `clk_d` input. The `almost_full_d` output indicates to the pop interface that the FIFO is almost full when there are no more than `af_level_d` empty locations in the FIFO as perceived by the pop interface.

The `af_level_d` input port defines the almost full threshold with respect to the entire FIFO of the pop interface independent of that of the push interface. The `almost_full_d` output is useful when it is desirable to pop data out of the FIFO in bursts (without allowing the FIFO to become empty).

### Property of almost\_full\_d

If `almost_full_d` is active (high) then there are at least  $(\text{eff\_depth} - \text{af\_level\_d})$  words in the FIFO, where `eff_depth` is defined in Table 1-2 on page 4. Therefore such status indicates that the pop interface can safely and unconditionally pop  $(\text{eff\_depth} - \text{af\_level\_d})$  words out of the FIFO. This property guarantees that such a “blind pop” operation will not underrun the FIFO.

## full\_d

The `full_d` output, active high, is synchronous to the `clk_d` input. `full_d` indicates to the pop interface that the FIFO is full. The action of popping the first word out of a full FIFO is controlled by the pop interface. Thus, the `full_d` output goes low at the rising edge of the `clk_d` that causes the first word to be popped.

The action of the last word being pushed into a nearly full FIFO is controlled by the push interface. This means the `full_d` output is asserted only after the new write pointer from the push interface is synchronized to `clk_d` and processed by the status flag state logic.

### Property of full\_d

If `full_d` is active (high) then the FIFO is truly full. This property does not apply to `full_s`.

## Blind Popping

*Blind popping* is the operation of performing consecutive-cycle popping without the risk of FIFO underruns (underflows). The number of consecutive popping cycles is predicated on the setting of level thresholds (`ae_level_d` and `af_level_d`) and the initiation of popping is based on the state of the destination domain status flags (`almost_empty_d` and `almost_full_d`, respectively, `half_full_d`, or `full_d`). In general, any popping operation must rely on one or more of the provided destination domain status flags (including `empty_d`) to know when popping can begin and to prevent underrun. A common practice for implementing blind popping operations is to use the `ae_level_d` input value coupled with the `almost_empty_d` status flag. For example, if the `ae_level_d` is set to 1 and the destination domain interface of the DW\_fifoctl\_2c\_df calculates a `word_cnt_d` of 2 and the pre-fetch cache has a sufficient number of words installed to guarantee contiguously popped words, then the registered `almost_empty_d` status flag will go from 1 to 0. Once the `almost_empty_d` is at 0, pop operations can immediately start (`pop_d_n` of logic 0 sampled on the next rising-edge of `clk_d`). The duration of consecutive popping cycles (or *blind popping*) to guarantee no FIFO underruns would then be 2 (`ae_level_d` + 1) after which the pop request *must* be released (for example, `pop_d_n` must go to logic 1).

**Note**

The word count output (`word_cnt_d`) cannot be used for triggering blind popping operations

## Simulation Methodology

Because this component contains synchronizing devices, there are two methods available for simulation. One method is to use the simulation models that emulate the RTL model. Or, you can enable modeling of random skew between bits of signals traversing to and from each domain (called missampling).

To use the simulation models that emulate the RTL model, no special configuration is required.

To use missampling requires the following considerations:

- To enable missampling in Verilog simulations, define the macro DW\_MODEL\_MISSAMPLES:

```
`define DW_MODEL_MISSAMPLES
```

If `DW_MODEL_MISSAMPLES` is defined, the `verif_en` parameter comes into play to configure the simulation model as described by [Table 1-3](#) on page 5. If `DW_MODEL_MISSAMPLES` is not defined, the Verilog simulation model behaves as if `verif_en` is set to 0.

- To enable missampling in VHDL simulations, a simulation architecture named `sim_ms` is provided. The parameter `verif_en` only has meaning when using `sim_ms`. That is, when the `sim` simulation architecture is used instead, the model behaves as though `verif_en` is set to 0. For examples of how each architecture is used, see [“HDL Usage Through Component Instantiation - VHDL”](#) on page 44.

## Suppressing Warning Messages During Verilog Simulation

The Verilog simulation model includes macros that allow you to suppress warning messages during simulation.

To suppress all warning messages for all DWBB components, define the DW\_SUPPRESS\_WARN macro in either of the following ways:

- Specify the Verilog preprocessing macro in Verilog code:
 

```
`define DW_SUPPRESS_WARN
```
- Or, include a command line option to the simulator, such as:
 

```
+define+DW_SUPPRESS_WARN (which is used for the Synopsys VCS simulator)
```

The warning messages for this model include the following:

- If values other than 1 or 0 are present on a clock port, the following message is displayed:

```
WARNING: <instance_path>.<clock_name>_monitor:
         at time = <timestamp>, Detected unknown value, x, on <clock_name> input.
```

To suppress only this warning message for all DWBB components, use the following macro:

- Define the DW\_DISABLE\_CLK\_MONITOR macro. You can define this macro in the following ways:
  - Specify the Verilog preprocessing macro in Verilog code:
 

```
`define DW_DISABLE_CLK_MONITOR
```
  - Or, include a command line option to the simulator, such as:
 

```
+define+DW_DISABLE_CLK_MONITOR (which is used for the Synopsys VCS simulator)
```

This message is also suppressed using the DW\_SUPPRESS\_WARN macro explained earlier.

## Simulation Assertions (SystemVerilog only)

The Verilog simulation model incorporates SystemVerilog assertions covering functionality in both clock domains. By default, all the assertions have a reporting severity of error but they can be changed to report as warning, fatal, or not at all by defining the preprocessing variable named DW\_SVA\_MODE. Not defining DW\_SVA\_MODE behaves as if it were defined as 2 (report as error). So, for example, if the desire is to have all the assertions in this component report with severity of warning do the following:

```
`define DW_SVA_MODE 1
```

See [Table 1-6](#) on page 8 for the assertion reporting severity based on the DW\_SVA\_MODE value. It is important to note the value of DW\_SVA\_MODE determines the same reporting severity of all the assertions within this component.

**Table 1-11 DW\_SVA\_MODE value and Assertion Severity**

DW_SVA_MODE value	Assertion Report Severity
not defined (default)	error

**Table 1-11 DW\_SVA\_MODE value and Assertion Severity (Continued)**

DW_SVA_MODE value	Assertion Report Severity
0	disable reporting
1	warning
2	error
3	fatal

The following is a list of some (but not all) of the assertions included in this component:

- Word counts (`fifo_word_cnt_s`, `word_cnt_s`, `word_cnt_d`) are legal values
- Status flags (full and empty in both clock domains) are in the proper state under system reset and clearing conditions.
- Full and empty status are in the proper states when not in system reset or clearing conditions
- Internal addresses and pointers do not change under push and/or pop error cases.

**Note**

For assertions related to word counts, initiation of system reset in one domain could result, temporarily, in the pointer math in the other domain to calculate values outside of the acceptable depth of the RAM or FIFO. However, after reset conditions, namely after both domains have been initialized and held idle as described in [“Reset Considerations”](#) on page 10 the word counts will settle into their allowed ranges.

## Timing Waveforms

[Figure 1-6](#) on page 31 shows an eight-deep RAM configured with `mem_mode = 0`. The `mem_mode` setting implies that there is a one deep cache in the destination domain and, hence, defines a nine-deep FIFO. With the source clock (`clk_s`) slower by a factor of two compared to the destination clock (`clk_d`), nine consecutive pushes (`push_s_n` asserted) can be issued without overrunning the RAM since the destination interface is able to read from RAM at least one data packet.

Notice that after each `clk_s` cycle that samples `push_s_n` as 0, `word_cnt_s[3:0]` increments (`word_cnt_s[3:0]` represents the number of valid entries in RAM). The only exception is on the sixth push in which `word_cnt_s[3:0]` holds at a value of 5. This is due to the cache in the destination getting load with the first location in the RAM. Thus, freeing up that location and reducing the number of valid entries. So, if the sixth push did not happen, the `word_cnt_s[3:0]` would have been 4 due to the loading of the first RAM location to cache (which increments RAM read address from 0 to 1). But with the sixth push occurring where it did, the value of `word_cnt_s[3:0]` stays at 5.

Furthermore, during the pushing activity the destination domain logic detects that the cache is empty and the RAM becomes non empty as indicated by the signal `word_cnt_d[3:0]` registering a value 1 (non-zero value). This detection initiates a read operation of RAM.

After nine consecutive pushes are performed, a tenth push is issued that causes an overrun condition of the RAM indicated by `error_s` going to 1. The `error_s` signal only stays at 1 for a single `clk_s` cycle in this case because the `err_mode` parameter is set to 1.

**Attention**

The signal, Actual\_FIFO\_WordCnt[3:0], at the bottom of the waveform is provided only as reference and it does not exist in the DW\_fifo\_2c\_df component.  
Actual\_FIFO\_WordCnt[3:0] is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

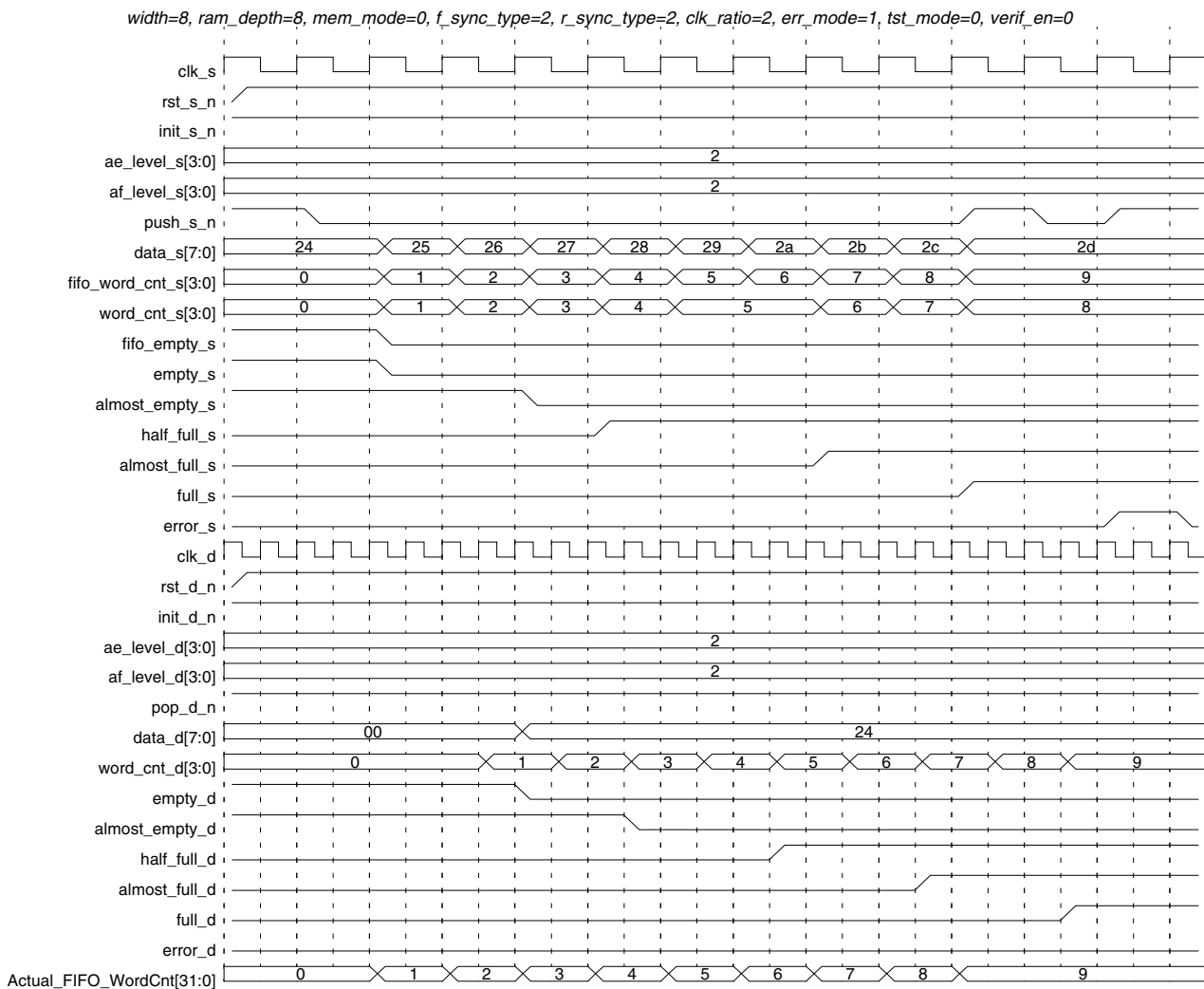
**Figure 1-6 Push Until Full and Error with clk\_s Slower than clk\_d (RAM Depth Is a Power of 2)**

Figure 1-7 shows popping activity from the FIFO full condition to the FIFO empty condition. As long as `empty_d` is a 0, `pop_d_n` is asserted (low). `word_cnt_d[3:0]` represents the number of valid entries in the FIFO. In this example, before popping is performed the `word_cnt_d[3:0]` value is 9 and `empty_d` is 0. Therefore, popping nine consecutive `clk_d` cycles empties the FIFO as shown in the waveform.

A tenth pop is performed when `empty_d` is 1, which causes an underrun of the cache and the `error_d` status signal to go high for a `clk_d` cycle. The `error_d` signal is only active for one `clk_d` cycle because the parameter `err_mode` is set to 1.

NOTE: The signal, `Actual_FIFO_WordCnt[3:0]`, at the bottom of the waveform is provided only as reference and it does not exist in the `DW_fifo_2c_df` component. `Actual_FIFO_WordCnt[3:0]` is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

**Figure 1-7 Pop Until Empty and Error with `clk_s` Slower than `clk_d` (RAM Depth Is a Power of 2)**

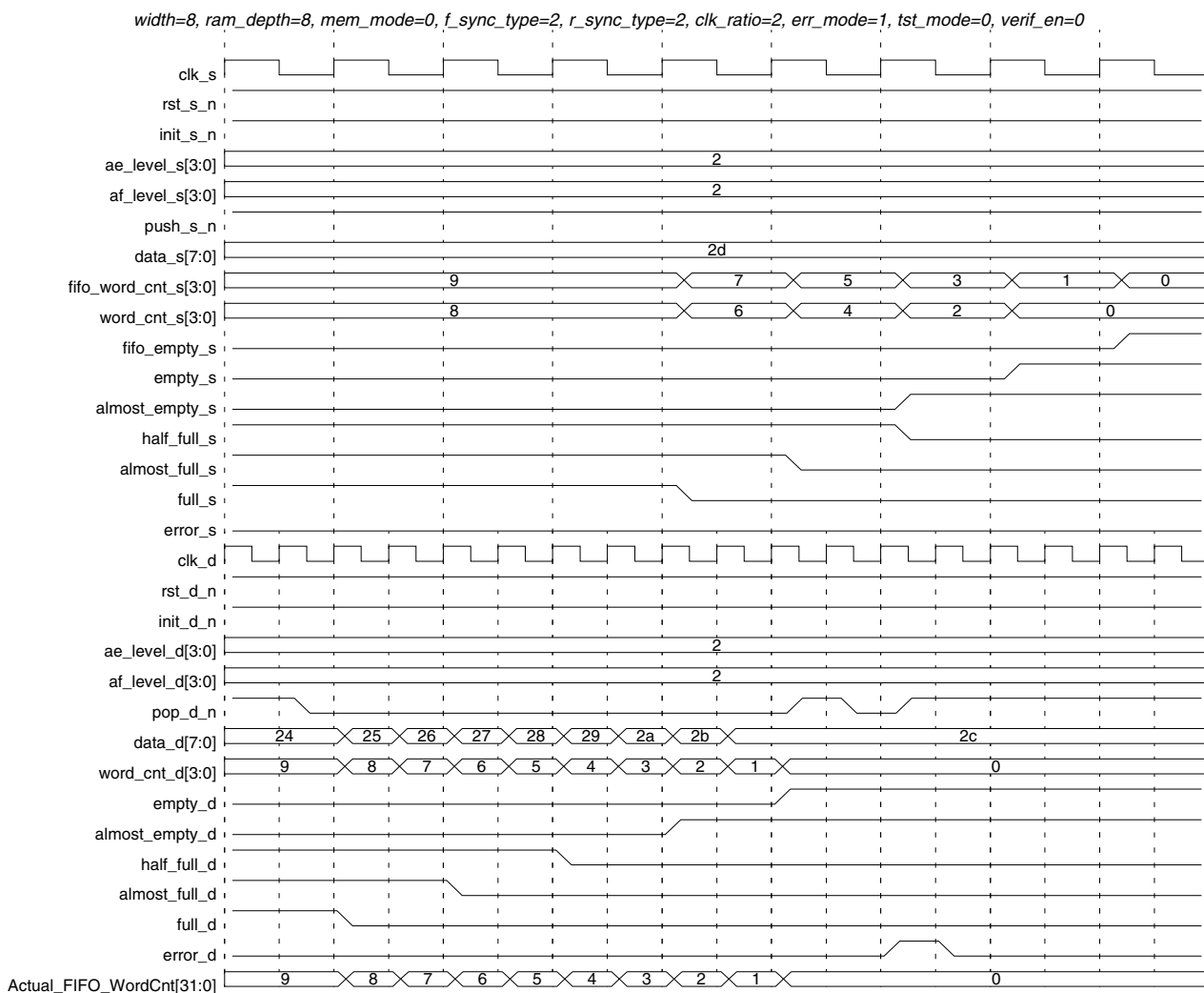




Figure 1-8 shows the latency of a single word push and pop along with the empty flags for both domains with *f\_sync\_type* and *r\_sync\_type* parameters set to 2.

**Figure 1-8 Single Word Push/Pop Timing with Double Synchronization**

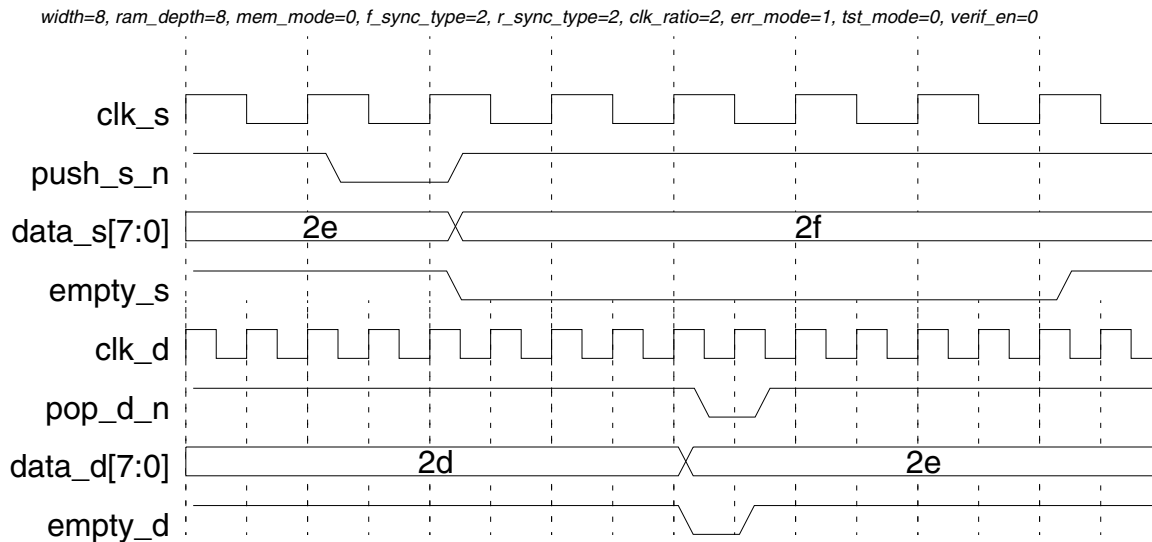


Figure 1-9 is similar to Figure 1-6 on page 31 except that the parameters *ram\_depth*, *mem\_mode*, and *err\_mode* have been changed.

With the combination of *ram\_depth* being 6 and *mem\_mode* set to 3, the FIFO depth derives to be nine. The *mem\_mode* setting of 3 indicates that the RAM that is being used in the DW\_fifo\_2c\_df contains re-timing of read address and control signals and re-timing of the RAM data output. Having RAM configure this way requires a cache depth of three (automatically derived). So, with *ram\_depth* being 6 and the cache depth being three, the overall FIFO depth is nine.

The parameter *err\_mode* in this timing diagram is set to 1. Therefore, in the event of the overrun condition of the RAM, the *error\_s* status flag only asserts upon occurrence of the error. If no overrun condition is present, then *error\_s* de-asserts on the next clock cycle.

NOTE: The signal, *Actual\_FIFO\_WordCnt*[3:0], at the bottom of the waveform is provided only as reference and it does not exist in the DW\_fifo\_2c\_df component. *Actual\_FIFO\_WordCnt*[3:0] is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

**Figure 1-9 Push Until RAM Full and Error (RAM Depth Is Not a Power of 2)**

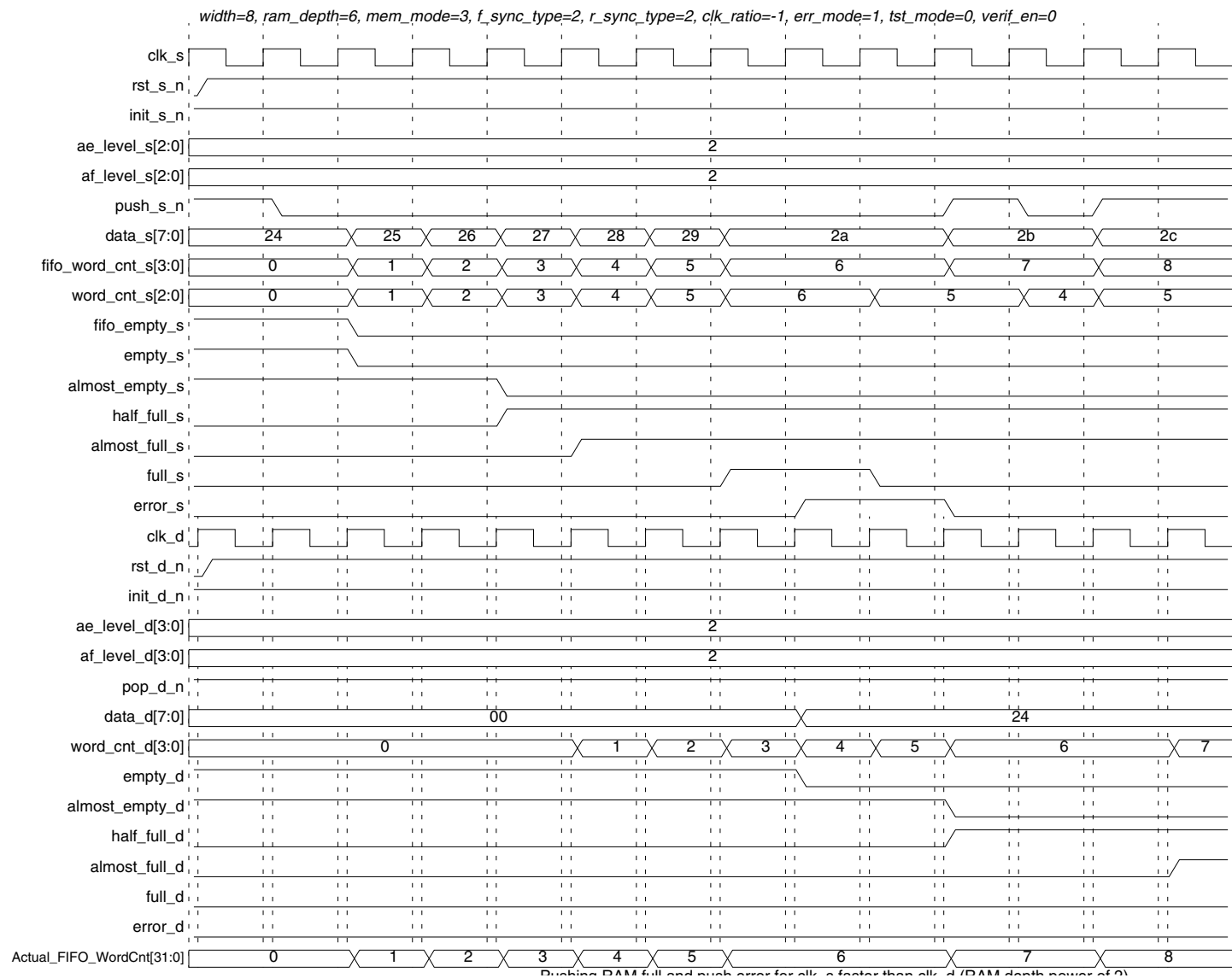


Figure 1-10 is similar to Figure 1-7 on page 32 except that the parameters *ram\_depth*, *mem\_mode*, and *err\_mode* have been changed. With the combination of *ram\_depth* being 6 and *mem\_mode* set to 3, the FIFO depth derives to be nine. The *mem\_mode* setting of 3 indicates that the RAM that is being used in the DW\_fifo\_2c\_df contains re-timing of read address and control signals and re-timing of the RAM data output. Having RAM configure this way requires a cache depth of three (automatically derived). So, with *ram\_depth* being 6 and the cache depth being three, the overall FIFO depth is nine.

The parameter *err\_mode* in this timing diagram is set to 0. Therefore, in the event of the overrun condition of the FIFO, the *error\_d* status flag asserts and stays asserted until either a system reset or coordinated clearing sequence of the two domains.



### Attention

The signal, Actual\_FIFO\_WordCnt[3:0], at the bottom of the waveform is provided only as reference and it does not exist in the DW\_fifo\_2c\_df component. Actual\_FIFO\_WordCnt[3:0] is meant to provide an idealized value of the number valid entries in the FIFO at any one time independent of either interface.

**Figure 1-10 Pop Until Empty and Error with *clk\_s* Slower than *clk\_d* (RAM Depth a Non-power of 2)**

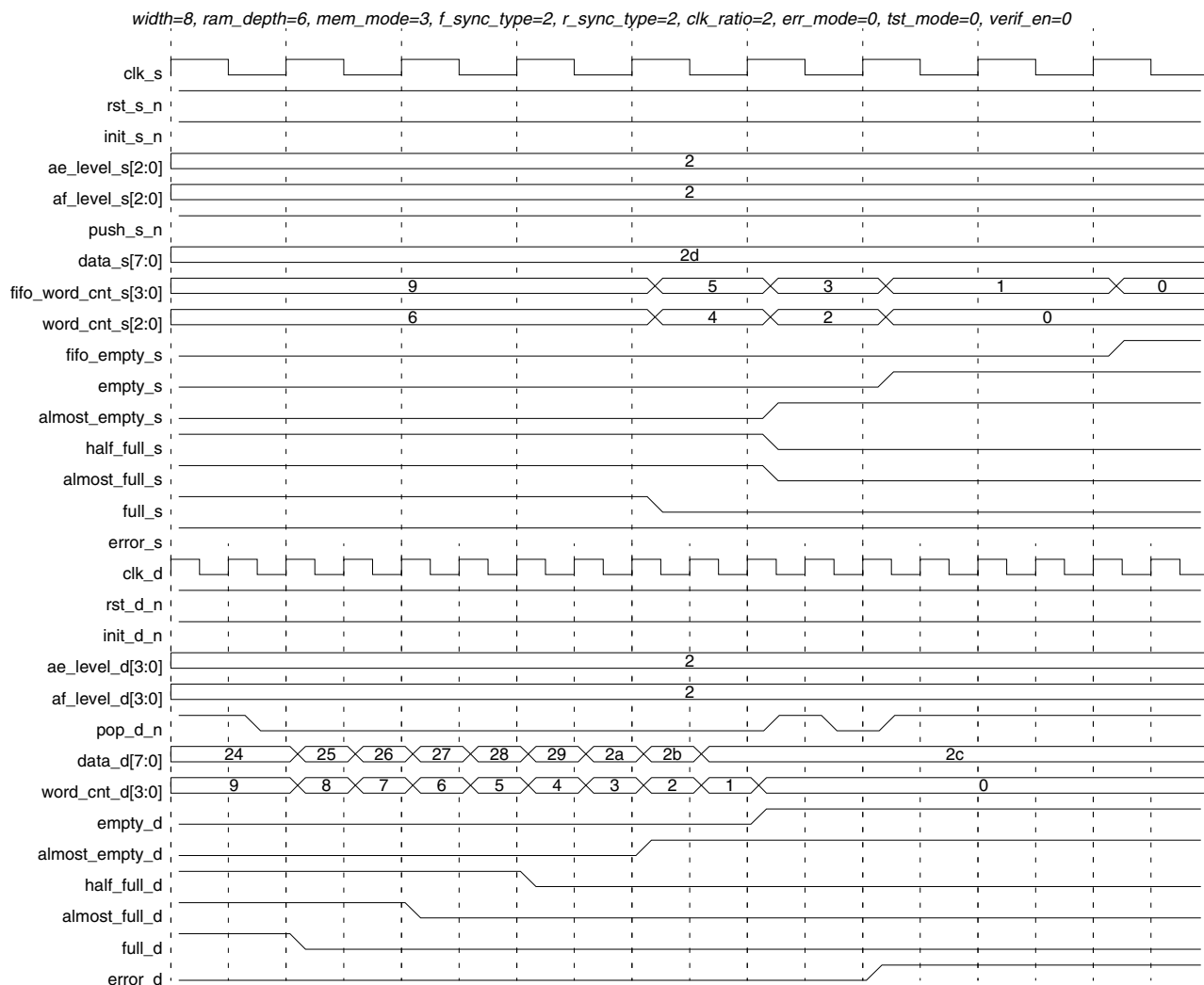


Figure 1-11 on page 37 shows the initiation of the coordinated clearing sequence from the source domain via assertion of `clr_s`. In this case `clr_s` is a single `clk_s` cycle, but the length of `clr_s` assertions are not restricted. The clearing-related signals are grouped at the bottom of the Push Interface and Pop Interface signal groups in the timing diagram.

When `clr_s` is asserted it gets synchronized at the destination domain (based on *f\_sync\_type*) which activates the `clr_in_prog_d`. `clr_in_prog_d` is useful for destination sequential logic in that it can be used to initialize circuits knowing that the source domain is not scheduled to push any data packets until the clearing sequence is complete. The event that produces the `clr_in_prog_d` assertion is then fed back to the source domain where it is synchronized (based on *r\_sync\_type*) to generate the `clr_sync_s` pulse. On the heels of the `clr_sync_s` pulse, the `clr_in_prog_s` signal gets activated. Similar to the `clr_in_prog_d` signal, `clr_in_prog_s` and/or `clr_sync_s` can be used to initialize source domain sequential logic since it's implied that no destination domain popping will be occurring until the clearing sequence is completed.

The `clr_sync_s` event is then sent back for synchronization in the destination domain to de-activate `clr_in_prog_d` and generate the `clr_cmplt_d` indicating to the destination domain that the source domain has been cleared and it can be in the “waiting” state for popping.

Now that the destination domain perceives that its clear sequence is done, that event is sent back to the source domain for synchronization which, in turn, de-activates `clr_in_prog_s` and produces a `clr_cmplt_s` pulse. The de-activation of `clr_in_prog_s` and subsequent `clr_cmplt_s` pulse indicates to the source domain that the destination domain logic has been cleared and all is ready for more pushing of data.

Figure 1-11 clr\_s Initiated Clearing Sequence

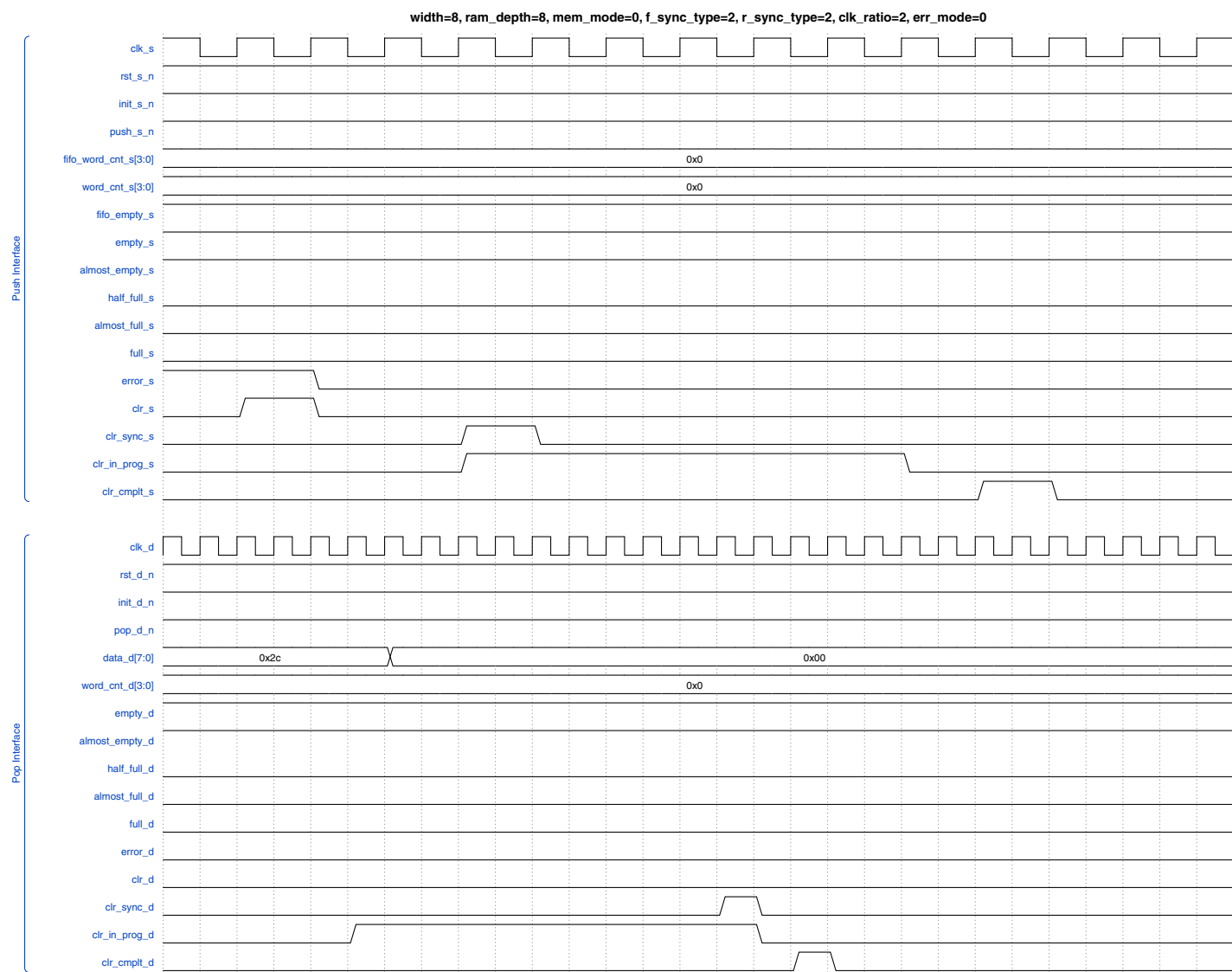


Figure 1-12 on page 38 describes the timing from an initiated `clr_d` pulse. The `clr_d` initiated clearing sequence is similar to the `clr_s` initiated clearing sequence with fewer synchronization feedback paths from beginning to completion.

**Figure 1-12 `clr_d` Initiated Clearing Sequence**

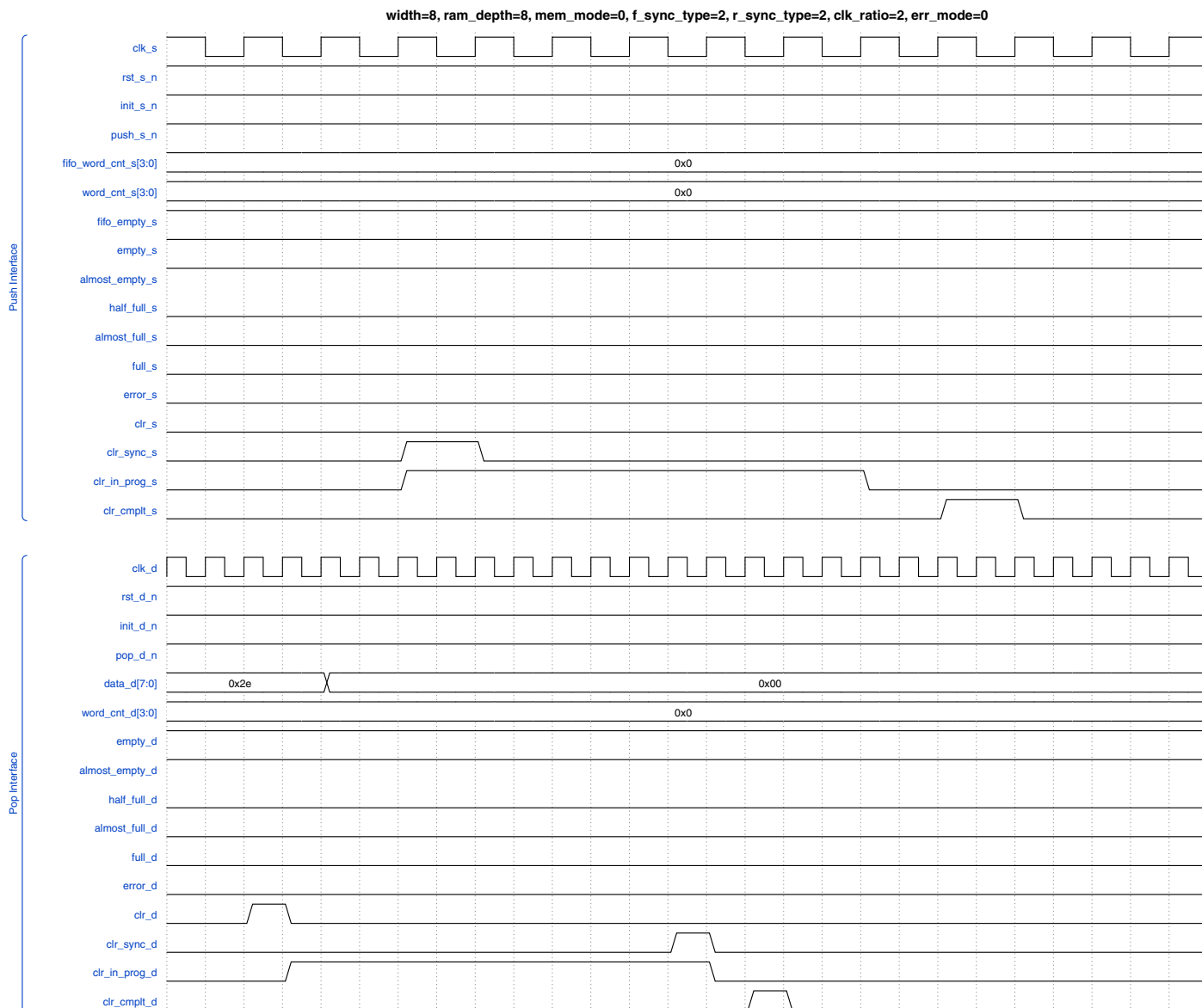


Figure 1-13 shows an initiation of a `clr_s` where its duration is much longer than one `clk_s` cycle. From this, the behavior of the `clr_in_prog_s` and `clr_in_prog_d` flags are sustained longer than those seen in Figure 1-11 on page 37 in which `clr_s` was only asserted a single `clk_s` cycle.

**Figure 1-13 Initiation of `clr_s` Much Longer than `clk_s` cycle**

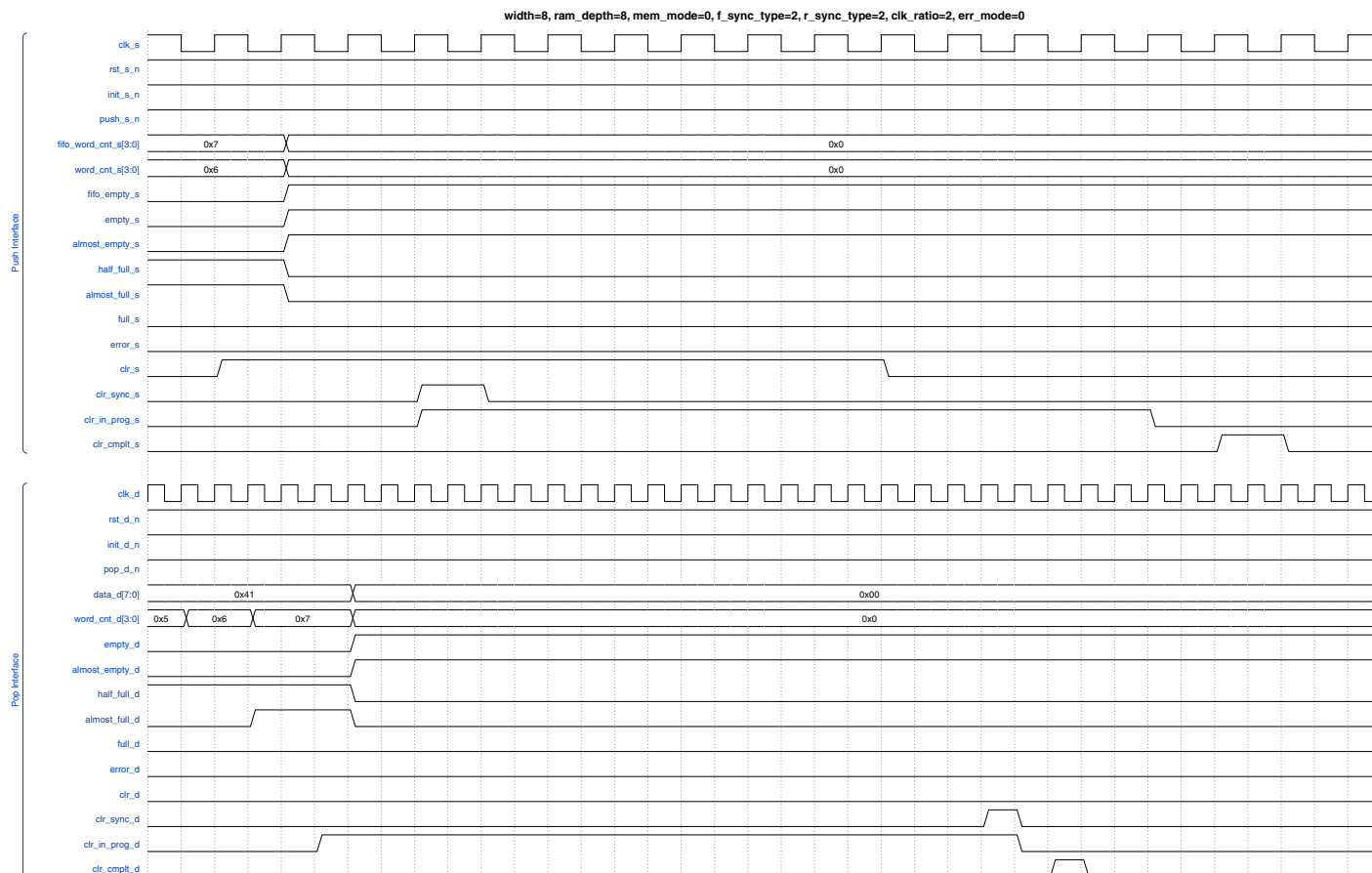


Figure 1-14 shows an example of a sequence where `rst_s_n` and `rst_d_n` are asserted. In this case, `clk_d` is faster than `clk_s` with `rst_d_n` asserted first followed by the assertion of `rst_s_n` within "`r_sync_type + 1`" `clk_s` cycles (`r_sync_type` is 2 in this example). Note that the word counts and addresses go to 0 and the status flags settle into the appropriate state.

**Figure 1-14 Example of Asynchronous System Reset**

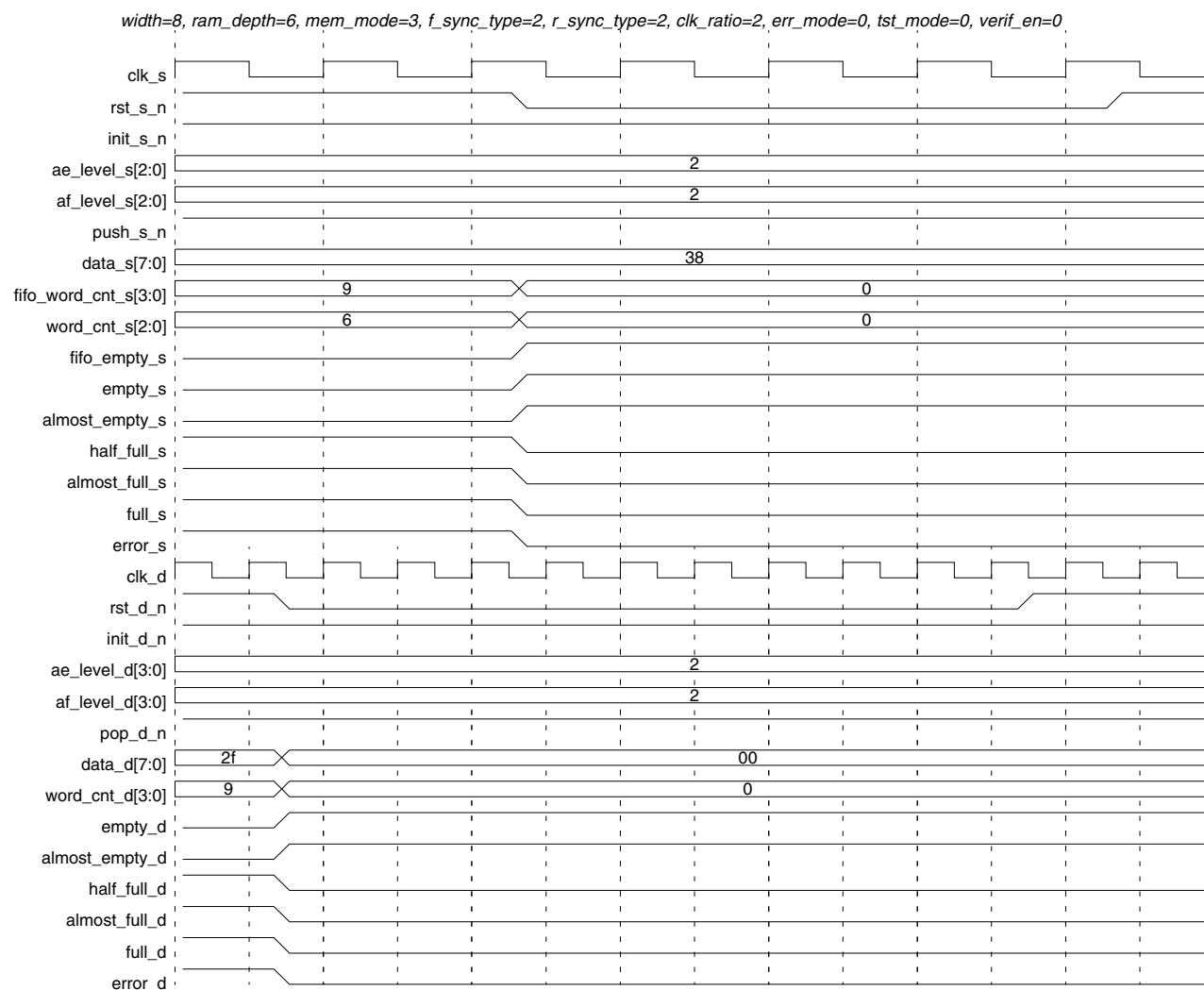
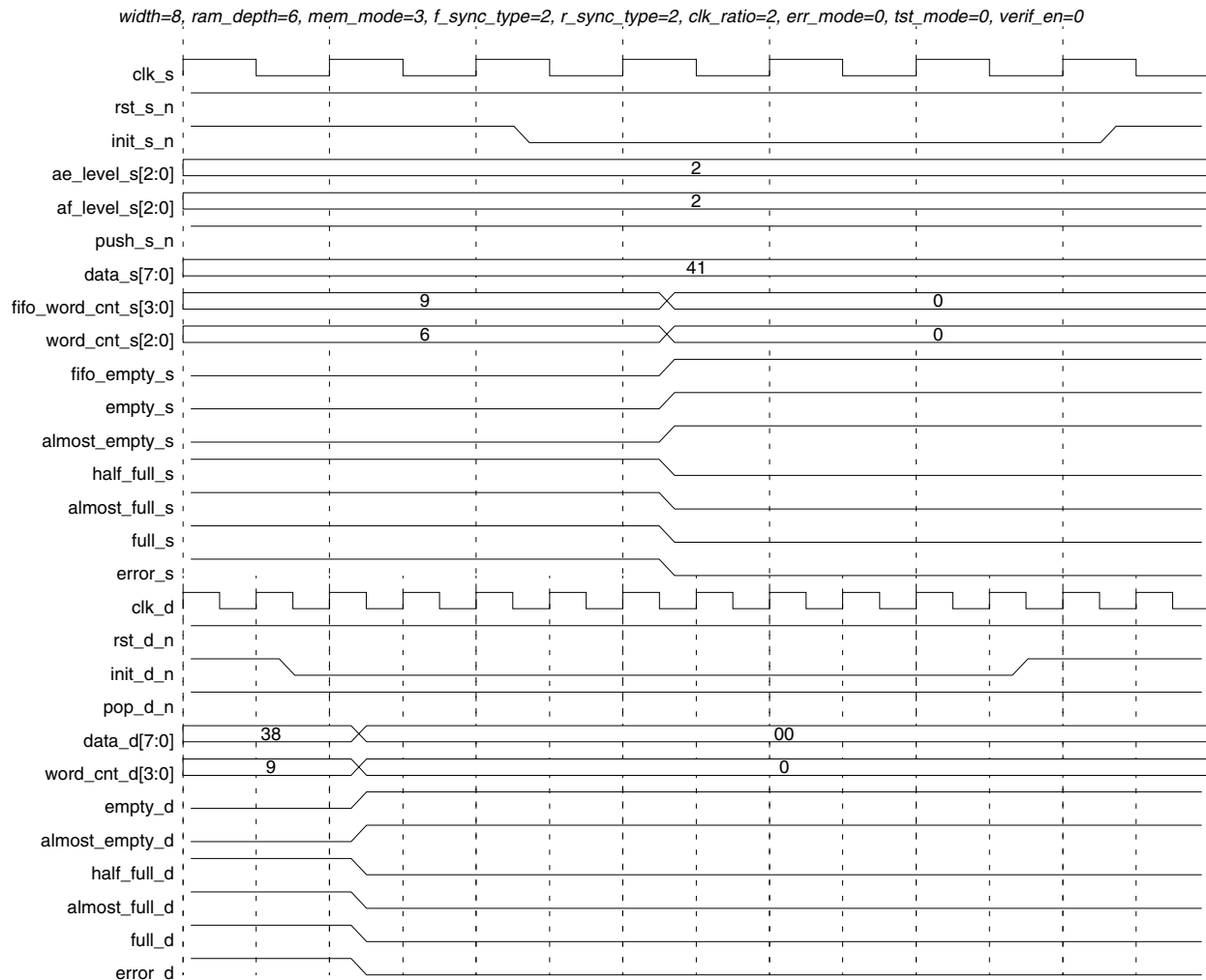




Figure 1-15 shows an example of a sequence where `init_s_n` and `init_d_n` are asserted. In this case, `clk_d` is faster than `clk_s` with `init_d_n` asserted first followed by the assertion of `init_s_n` within “`r_sync_type + 1`” `clk_s` cycles (`r_sync_type` is 2 in this example). Note that the word counts and addresses go to 0 and the status flags settle into the appropriate state.

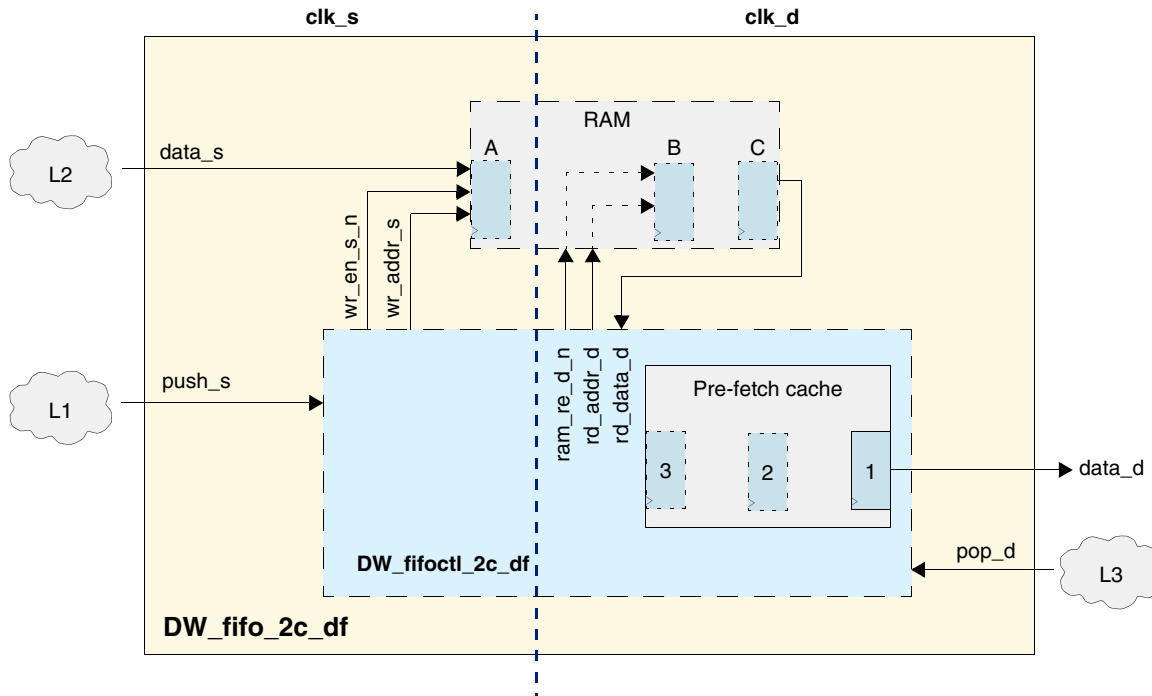
**Figure 1-15 Example of Synchronous System Reset**



## Considerations for Setting the mem\_mode Parameter

Depending on the logic being supplied to the DW\_fifo\_2c\_df, the setting of the *mem\_mode* parameter is determined. Refer to [Figure 1-16](#) for the discussion that follows.

**Figure 1-16 mem\_mode Settings Based on System Design**



If the contains L1 and/or L2 logic clouds or delays that cause either *push\_s* and/or *data\_s*, respectively, to be late-arriving, then there potentially would be the need to re-time the writing signals into the RAM as denoted by register A. If register A of the RAM is required in the design, the *mem\_mode* parameter should be set to 4 or greater depending on the existence of registers B and C in the RAM.

Similarly, if there is an L3 logic cloud or delays that cause *pop\_d* to be late-arriving enough to require a register B in the RAM, then *mem\_mode* should be set to 2, 3, 6, or 7 depending on the existence of registers A and C in the RAM.

If delay is minimal through L1, L2, and L3 which does not require either register A or B in the RAM, then depending on the internal delay of the datapath through the RAM to its output a re-timing register C may or may not be needed. If needed, then the *mem\_mode* must be set to either 1 (register C is exists) or 0 (register C does not exists).

The following is a table that lists the all possible supported RAM architectures and the required *mem\_mode* setting along with the resulting structure of the pre-fetch cache.

**Table 1-12 RAM Configuration Determines mem\_mode Setting**

Does RAM register exist?			<i>mem_mode</i> Setting	Pre-fetch Cache Structure
A	B	C		
no	no	no	0	register 1
no	no	yes	1	register 1 and 2
no	yes	no	2	register 1 and 2
no	yes	yes	3	register 1, 2 and 3
yes	no	no	4	register 1
yes	no	yes	5	register 1 and 2
yes	yes	no	6	register 1 and 2
yes	yes	yes	7	register 1, 2, and 3

## Enabling minPower

You can instantiate this component without enabling minPower, but to achieve power savings from the low-power implementation “lpwr” (see [Table 1-4](#) on page 7), you must enable minPower optimization, as follows:

- Design Compiler

- Version P-2019.03 and later:

```
set power_enable_minpower true
```

- Before version P-2019.03 (requires the DesignWare-LP license feature):

```
set synthetic_library {dw_foundation.sldb dw_minpower.sldb}
set link_library {* $target_library $synthetic_library}
```

- Fusion Compiler

Optimization for minPower is enabled as part of the total\_power metric setting. To enable the total\_power metric, use the following:

```
set_qor_strategy -stage synthesis -metric total_power
```

## Related Topics

- [Memory – FIFO Overview](#)
- [DesignWare Building Block IP User Guide](#)

## HDL Usage Through Component Instantiation - VHDL

```

library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_Foundation_comp.all;

entity DW_fifo_2c_df_inst is
    generic (
        inst_width           : POSITIVE := 8;
        inst_ram_depth       : POSITIVE := 8;
        inst_mem_mode        : NATURAL  := 5;
        inst_f_sync_type     : NATURAL  := 2;
        inst_r_sync_type     : NATURAL  := 2;
        inst_clk_ratio       : INTEGER  := 7;
        inst_rst_mode        : NATURAL  := 1;
        inst_err_mode        : NATURAL  := 0;
        inst_tst_mode        : NATURAL  := 0;
        inst_verif_en        : NATURAL  := 2;
        inst_clr_dual_domain : NATURAL  := 1;
        inst_arch_type       : NATURAL  := 0
    );
    port (
        inst_clk_s           : in std_logic;
        inst_rst_s_n         : in std_logic;
        inst_init_s_n        : in std_logic;
        inst_clr_s           : in std_logic;
        inst_ae_level_s      : in std_logic_vector
                               (bit_width(inst_ram_depth+1)-1 downto 0);
        inst_af_level_s      : in
                               std_logic_vector(bit_width(inst_ram_depth+1)-1 downto 0);
        inst_push_s_n        : in std_logic;
        inst_data_s          : in std_logic_vector(inst_width-1 downto 0);
        clr_sync_s_inst      : out std_logic;
        clr_in_prog_s_inst   : out std_logic;
        clr_cmplt_s_inst     : out std_logic;
        fifo_word_cnt_s_inst : out
                               std_logic_vector(bit_width(inst_ram_depth+(1+(inst_mem_mode
                               mod 2))+(inst_mem_mode/2 mod 2))-1 downto 0);
        word_cnt_s_inst      : out
                               std_logic_vector(bit_width(inst_ram_depth+1)-1 downto 0);
        fifo_empty_s_inst    : out std_logic;
        empty_s_inst         : out std_logic;
        almost_empty_s_inst  : out std_logic;
        half_full_s_inst     : out std_logic;
        almost_full_s_inst   : out std_logic;
        full_s_inst          : out std_logic;
        error_s_inst         : out std_logic;
        inst_clk_d           : in std_logic;
    );
end entity DW_fifo_2c_df_inst;

```

```

    inst_rst_d_n      : in std_logic;
    inst_init_d_n     : in std_logic;
    inst_clr_d        : in std_logic;
    inst_ae_level_d   : in
        std_logic_vector(bit_width(inst_ram_depth+1)-1 downto 0);
    inst_af_level_d    : in
        std_logic_vector(bit_width(inst_ram_depth+1)-1 downto 0);
    inst_pop_d_n      : in std_logic;
    clr_sync_d_inst   : out std_logic;
    clr_in_prog_d_inst : out std_logic;
    clr_cmplt_d_inst  : out std_logic;
    data_d_inst       : out std_logic_vector(inst_width-1 downto 0);
    word_cnt_d_inst   : out
        std_logic_vector(bit_width(inst_ram_depth+(1+(inst_mem_mode
        mod 2))+((inst_mem_mode/2) mod 2))-1 downto 0);
    empty_d_inst      : out std_logic;
    almost_empty_d_inst : out std_logic;
    half_full_d_inst  : out std_logic;
    almost_full_d_inst : out std_logic;
    full_d_inst       : out std_logic;
    error_d_inst      : out std_logic;
    inst_test         : in std_logic
);
end DW_fifo_2c_df_inst;

```

architecture inst of DW\_fifo\_2c\_df\_inst is  
begin

```

-- Instance of DW_fifo_2c_df
U1 : DW_fifo_2c_df
generic map ( width => inst_width, ram_depth => inst_ram_depth,
    mem_mode => inst_mem_mode, f_sync_type => inst_f_sync_type,
    r_sync_type => inst_r_sync_type, clk_ratio => inst_clk_ratio,
    rst_mode => inst_rst_mode, err_mode => inst_err_mode,
    tst_mode => inst_tst_mode, verif_en => inst_verif_en,
    clr_dual_domain => inst_clr_dual_domain,
    arch_type => inst_arch_type )
port map ( clk_s => inst_clk_s, rst_s_n => inst_rst_s_n, init_s_n =>
    inst_init_s_n,
    clr_s => inst_clr_s, ae_level_s => inst_ae_level_s, af_level_s =>
    inst_af_level_s,
    push_s_n => inst_push_s_n, data_s => inst_data_s, clr_sync_s =>
    clr_sync_s_inst,
    clr_in_prog_s => clr_in_prog_s_inst, clr_cmplt_s => clr_cmplt_s_inst,
    fifo_word_cnt_s => fifo_word_cnt_s_inst, word_cnt_s => word_cnt_s_inst,
    fifo_empty_s => fifo_empty_s_inst, empty_s => empty_s_inst,
    almost_empty_s => almost_empty_s_inst,
    half_full_s => half_full_s_inst, almost_full_s => almost_full_s_inst,
    full_s => full_s_inst, error_s => error_s_inst,

```

```
    clk_d => inst_clk_d, rst_d_n => inst_rst_d_n, init_d_n => inst_init_d_n,
    clr_d => inst_clr_d, ae_level_d => inst_ae_level_d, af_level_d =>
    inst_af_level_d,
    pop_d_n => inst_pop_d_n, clr_sync_d => clr_sync_d_inst,
    clr_in_prog_d => clr_in_prog_d_inst, clr_cmplt_d => clr_cmplt_d_inst,
    data_d => data_d_inst, word_cnt_d => word_cnt_d_inst,
    empty_d => empty_d_inst, almost_empty_d => almost_empty_d_inst,
    half_full_d => half_full_d_inst, almost_full_d => almost_full_d_inst,
    full_d => full_d_inst, error_d => error_d_inst, test => inst_test );

end inst;

-- Configuration for use with a VHDL simulator
-- pragma translate_off
library DW06;
configuration DW_fifo_2c_df_inst_cfg_inst of DW_fifo_2c_df_inst is
    for inst
        -- NOTE: If desiring to model missampling, uncomment the following
        -- line. Doing so, however, will cause inconsequential errors
        -- when analyzing or reading this configuration before synthesis.
        -- for U1 : DW_fifo_2c_df use configuration DW06.DW_fifo_2c_df_cfg_sim_ms; end
    for;
    end for; -- inst
end DW_fifo_2c_df_inst_cfg_inst;
-- pragma translate_on
```

## HDL Usage Through Component Instantiation - Verilog

```

module DW_fifo_2c_df_inst( inst_clk_s, inst_rst_s_n, inst_init_s_n,
    inst_clr_s, inst_ae_level_s, inst_af_level_s, inst_push_s_n,
    inst_data_s, clr_sync_s_inst, clr_in_prog_s_inst, clr_cmplt_s_inst,
    fifo_word_cnt_s_inst, word_cnt_s_inst, fifo_empty_s_inst,
    empty_s_inst, almost_empty_s_inst, half_full_s_inst,
    almost_full_s_inst, full_s_inst, error_s_inst, inst_clk_d,
    inst_rst_d_n, inst_init_d_n, inst_clr_d, inst_ae_level_d,
    inst_af_level_d, inst_pop_d_n, clr_sync_d_inst, clr_in_prog_d_inst,
    clr_cmplt_d_inst, data_d_inst, word_cnt_d_inst, empty_d_inst,
    almost_empty_d_inst, half_full_d_inst, almost_full_d_inst,
    full_d_inst, error_d_inst, inst_test );

parameter width = 8;
parameter ram_depth = 8;
parameter mem_mode = 5;
parameter f_sync_type = 2;
parameter r_sync_type = 2;
parameter clk_ratio = 3;
parameter rst_mode = 1;
parameter err_mode = 1;
parameter tst_mode = 0;
parameter verif_en = 1;
parameter clr_dual_domain = 1;
parameter arch_type = 0;
`define ram_cnt_width  4 // ceil(log2(ram_depth+1))
`define fifo_cnt_width 4 // ceil(log2((ram_depth+1+(mem_mode % 2)+((mem_mode/2) %
2))+1))

input inst_clk_s;
input inst_rst_s_n;
input inst_init_s_n;
input inst_clr_s;
input [`ram_cnt_width-1:0] inst_ae_level_s;
input [`ram_cnt_width-1:0] inst_af_level_s;
input inst_push_s_n;
input [width-1:0] inst_data_s;

output clr_sync_s_inst;
output clr_in_prog_s_inst;
output clr_cmplt_s_inst;
output [`fifo_cnt_width-1:0] fifo_word_cnt_s_inst;
output [`ram_cnt_width-1:0] word_cnt_s_inst;
output fifo_empty_s_inst;
output empty_s_inst;
output almost_empty_s_inst;
output half_full_s_inst;

```



```

output almost_full_s_inst;
output full_s_inst;
output error_s_inst;

input inst_clk_d;
input inst_rst_d_n;
input inst_init_d_n;
input inst_clr_d;
input [`fifo_cnt_width-1:0] inst_ae_level_d;
input [`fifo_cnt_width-1:0] inst_af_level_d;
input inst_pop_d_n;

output clr_sync_d_inst;
output clr_in_prog_d_inst;
output clr_cmplt_d_inst;
output [width-1:0] data_d_inst;
output [`fifo_cnt_width-1:0] word_cnt_d_inst;
output empty_d_inst;
output almost_empty_d_inst;
output half_full_d_inst;
output almost_full_d_inst;
output full_d_inst;
output error_d_inst;

input inst_test;

// Instance of DW_fifo_2c_df
DW_fifo_2c_df #(width, ram_depth, mem_mode, f_sync_type, r_sync_type, clk_ratio,
rst_mode, err_mode, tst_mode, verif_en, clr_dual_domain, arch_type)
  U1 ( .clk_s(inst_clk_s), .rst_s_n(inst_rst_s_n),
.init_s_n(inst_init_s_n), .clr_s(inst_clr_s), .ae_level_s(inst_ae_level_s),

.af_level_s(inst_af_level_s), .push_s_n(inst_push_s_n), .data_s(inst_data_s),
.clr_sync_s(clr_sync_s_inst), .clr_in_prog_s(clr_in_prog_s_inst),
.clr_cmplt_s(clr_cmplt_s_inst), .fifo_word_cnt_s(fifo_word_cnt_s_inst),
.word_cnt_s(word_cnt_s_inst), .fifo_empty_s(fifo_empty_s_inst),
.empty_s(empty_s_inst), .almost_empty_s(almost_empty_s_inst),
.half_full_s(half_full_s_inst), .almost_full_s(almost_full_s_inst),
.full_s(full_s_inst), .error_s(error_s_inst), .clk_d(inst_clk_d),
.rst_d_n(inst_rst_d_n), .init_d_n(inst_init_d_n), .clr_d(inst_clr_d),
.ae_level_d(inst_ae_level_d), .af_level_d(inst_af_level_d),
.pop_d_n(inst_pop_d_n), .clr_sync_d(clr_sync_d_inst),
.clr_in_prog_d(clr_in_prog_d_inst), .clr_cmplt_d(clr_cmplt_d_inst),
.data_d(data_d_inst), .word_cnt_d(word_cnt_d_inst), .empty_d(empty_d_inst),
.almost_empty_d(almost_empty_d_inst), .half_full_d(half_full_d_inst),
.almost_full_d(almost_full_d_inst), .full_d(full_d_inst),
.error_d(error_d_inst), .test(inst_test) );

```

endmodule

## Revision History

For notes about this release, see the [DesignWare Building Block IP Release Notes](#).

For lists of both known and fixed issues for this component, refer to the [STAR report](#).

For a version of this datasheet with visible change bars, click [here](#).

Date	Release	Updates
July 2022	DWBB_202203.3	<ul style="list-style-type: none"> <li>Adjusted waveforms for the <code>clr_s</code>-initiated clearing sequence in <a href="#">Figure 1-11</a> on page 37</li> <li>Adjusted waveforms for the <code>clr_d</code>-initiated clearing sequence in <a href="#">Figure 1-12</a> on page 38 and removed some description text that is no longer relevant</li> <li>Adjusted waveforms for a clearing sequence with the duration of <code>clr_s</code> much longer than one <code>clk_s</code> cycle in <a href="#">Figure 1-13</a> on page 39</li> </ul>
January 2022	DWBB_202106.5	<ul style="list-style-type: none"> <li>In the description of the waveforms in <a href="#">Figure 1-11</a> on page 37, removed text describing behavior that is no longer relevant</li> <li>In the description of the waveforms in <a href="#">Figure 1-12</a> on page 38, removed text describing behavior that is no longer relevant</li> </ul>
December 2021	DWBB_202106.4	<ul style="list-style-type: none"> <li>Adjusted details about <code>clr_sync_s</code> and <code>clr_sync_d</code> in the block diagram in <a href="#">Figure 1-1</a> on page 9</li> <li>Corrected the title of <a href="#">Figure 1-9</a> on page 34</li> <li>Adjusted waveforms for <code>clr_s</code>-initiated clearing sequence in <a href="#">Figure 1-11</a> on page 37</li> <li>Adjusted waveforms for <code>clr_d</code>-initiated clearing sequence in <a href="#">Figure 1-12</a> on page 38</li> <li>Adjusted waveforms for <code>clr_s</code> with duration much longer than one <code>clk_s</code> cycle in <a href="#">Figure 1-13</a> on page 39</li> </ul>
September 2020	DWBB_202009.0	<ul style="list-style-type: none"> <li>Clarified the width description for pins in <a href="#">Table 1-1</a> on page 2 that use “eff_depth” to calculate their width</li> <li>Corrected width specification for <code>test</code> input in <a href="#">Table 1-1</a> on page 2</li> <li>Corrected the waveforms in <a href="#">Figure 1-2</a> on page 16 and <a href="#">Figure 1-3</a> on page 17</li> <li>Clarified language about the number of stages in the component in “f_sync_type and r_sync_type” on page 18</li> <li>Clarified the usage of the <code>lpwr</code> implementation in <a href="#">Table 1-6</a> on page 8</li> </ul>
July 2020	DWBB_201912.5	<ul style="list-style-type: none"> <li>Adjusted the material in “Simulation Methodology” on page 28</li> <li>Adjusted content and title of “Suppressing Warning Messages During Verilog Simulation” on page 29 and added the <code>DW_SUPPRESS_WARN</code> macro</li> <li>Added the <code>arch_type</code> parameter to the examples on <a href="#">page 44</a> and <a href="#">page 47</a></li> </ul>
October 2019	DWBB_201903.5	<ul style="list-style-type: none"> <li>Added the “Disabling Clock Monitor Messages” section</li> </ul>

Date	Release	Updates
March 2019	DWBB_201903.0	<ul style="list-style-type: none"><li>■ Clarified license requirements in <a href="#">Table 1-4</a> on page <a href="#">7</a></li><li>■ Added <a href="#">“Enabling minPower”</a> on page <a href="#">43</a></li></ul>
October 2018	DWBB_201806.3	<ul style="list-style-type: none"><li>■ Removed the section “Memory Depth Considerations and Setting ram_depth” because it does not apply</li><li>■ Added this Revision History table and the document links on this page</li></ul>

## Copyright Notice and Proprietary Information

© 2022 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

### Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

### Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

### Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>.

All other product or company names may be trademarks of their respective owners.

### Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

### Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.  
[www.synopsys.com](http://www.synopsys.com)